A MANUAL OF GIER ALGOL

as developed by

Henning Christensen, Jørn Jensen, Peter Kraft, Paul Lindgreen,
Peter Naur, Knut-Sivert Skog and Peter Villemoes.

First edition

Edited by Peter Naur

REGNECENTRALEN, COPENHAGEN

1963

CONTENTS.

The ALGOL 60 Report.

Throughout the present Manual reference is made to the ALGOL 60 Report or the Revised ALGOL 60 Report. The differences between these two documents are slight and do not influence the numbering of sections. The full references of these reports are as follows:

J. W. Backus, et. al., Report on the Algorithmic Language ALGOL 60 (ed. P. Naur), Numerische Mathematik 2 (1960), pp. 106-136; Acta Polytechnica Scandinavica: Math. And Comp. Mach. Ser. no. 5 (1960); Comm. ACM 3 no. 5 (1960), pp. 299-314.

J. W. Backus, et. al., Revised Report on the Algorithmic Language ALGOL 60 (ed. P. Naur), Regnecentralen, Copenhagen (1962), Comm. ACM 6 no. 1 (1963), pp 1-17; Computer Journal 5 (1963), pp. 349-367; Numerische Mathematik (in press).

INTRODUCTION.

The decision that an ALGOL compiler for the GIER should be written was made in January 1962. The work was started almost immediately and in August 1962 a preliminary version of the compiler could be distributed to all GIER installations. This version was complete except for some standard input and output procedures. The first definitive version, which also corrected a number of errors found through the extensive practical use of the preliminary version, was distributed in February 1963.

Like its predecessor DASK ALGOL the GIER ALGOL language lies sufficiently close to the ALGOL 60 reference language to make it practical to use the ALGOL 60 Report directly as the basic manual. The exact specifications of GIER ALGOL are then defined through the set of corrections and additions of the ALGOL 60 Report given in the present Manual. Because of this intimate relation to the ALGOL 60 Report the numbering of sections within the present Manual have been chosen to be a direct continuation of the section numbers of the ALGOL 60 Report.

Since there is also a considerable interest in the differences between GIER ALGOL and DASK ALGOL the section numbering of the MANUAL OF THE DASK ALGOL LANGUAGE have been retained through the major part of the present Manual. Because of the greater generality of GIER ALGOL several sections have become empty through this precaution. The more important differences between the two Manuals touch the following sections: 6.5, 7, 8.1, 8.2, 8.3.1, 8.3.6, 8.4.1, 8.4.3, 8.8, 9.1, 9.2.3-9.2.6, 9.4.3.6, 9.5.3, 9.8-9.12, 10, 11, appendix 1-5.

The Manual was typed by Kirsten Andersen.

6. 8-CHANNEL PUNCH TAPE CODE AND FLEXOWRITER KEYBOARD.

6.1. PRINTED SYMBOLS.

Lower case	Upper case	Code	Lower case	Upper case	Code
a	A	oo . o	w	W	o .oo
b	B	oo . o	x	X	oo .ooo
c	C	ooo . oo	y	Y	ooo.
d	D	oo .o	z	Z	o o. o
e	E	ooo .o o	æ	Æ	ooo .
f	F	ooo .oo	ø	Ø	o oo. oo
g	G	oo .ooo	0	^	o .
h	H	oo o.	1	v	. o
i	I	oooo. o	2	x	. o
j	J	o o . o	3	/	o . oo
k	K	o o . o	4	=	.o
l	L	o . oo	5	;	o .o o
m	M	o o .o	6	[	o .oo
n	N	o .o o	7	]	.ooo
o	O	o .oo	8	(	o.
p	P	o o .ooo	9	)	oo. o
q	Q	o oo.	'	10	ooo. oo
r	R	o o. o	.	:	oo o. oo
s	S	oo . o	-	+	o .
t	T	o . oo	<	>	oo . o
u	U	oo .o	_	|	o.oo
v	V	o .o o			

The key for _| does not advance the carriage.

6.2. TYPOGRAPHICAL SYMBOLS.

LOWER CASE	oooo. o	UPPER CASE	oooo.o	SPACE	o .
CAR RET	o .	TAB	ooo.oo		

6.3. CONTROL SYMBOLS.

STOP CODE	o. oo	TAPE FEED	oooo.ooo	PUNCH ADRES	o .
PUNCH OFF	o o.ooo	PUNCH ON	o o.o	AUX CODE	o.o

PUNCH ADRES and AUX CODE insert their respective codes when depressed simultaneously with any other key.

6.4. FLEXOWRITER KEYBOARD.

START READ	STOP READ	PUNCH ADRES		AUX CODE	STOP CODE	TAPE FEED

TAB	PUNCH OFF	x 2	/ 3	= 4	; 5	[6	] 7	(8	) 9	^ 0	v 1	| _	PUNCH ON

	Q q	W w	E e	R r	T t	Y y	U u	I i	O o	P p	> <	CAR RET

LOWER CASE	A a	S s	D d	F f	G g	H h	J j	K k	L l	Æ æ	Ø ø	LOWER CASE

UPPER CASE	Z z	X x	C c	V v	B b	N n	M m	10 '	: .	+ -	UPPER CASE

SPACE

6.5. NUMERICAL REPRESENTATIONS.

In the following table the characters have been arranged according to the numerical equivalent of the hole combination (after removal of the parity check hole). The first column gives the decimal value of the character, the second and third columns give the lower and upper case character, respectively, and the fourth column contains a G in the cases where the character is available only in GIER, but not on the flexowriter

	LOWER	UPPER			LOWER	UPPER	
0	SPACE			32	-	+	
1	1	∨		33	j	J	
2	2	×		34	k	K	
3	3	/		35	l	L	
4	4	=		36	m	M	
5	5	:		37	n	N	
6	6	[		38	o	O	
7	7	]		39	p	P	
8	8	(		40	q	Q	
9	9	)		41	r	R	
10	(NOT USED)			42	(NOT USED)		
11	STOP CODE			43	ø	Ø	
12	END CODE			44	PUNCH ON		
13	å	Å	G	45	(NOT USED)		
14		|		46	(NOT USED)		
15	(NOT USED)			47	(NOT USED)		
16	0	^		48	æ	Æ	
17	<	>		49	a	A	
18	s	S		50	b	B	
19	t	T		51	c	C	
20	u	U		52	d	D	
21	v	V		53	e	E	
22	w	W		54	f	F	
23	x	X		55	g	G	
24	y	Y		56	h	H	
25	z	Z		57	i	I	
26	(NOT USED)			58	LOWER CASE		
27	,	10		59	.	:	
28	CLEAR CODE			60	UPPER CASE		
29	RED RIBBON		G	61	SUM CODE		
30	TAB			62	BLACK RIBBON		G
31	PUNCH OFF			63	TAPE FEED		
				64	CAR RET		

7. THE RELATION BETWEEN GIER ALGOL AND ALGOL 60.

7.1. BASIC SYMBOLS.

7.1.1. Single character symbols.
7.1.1.1. Letters and digits. GIER ALGOL adds the letters
æ Æ ø Ø
to the reference alphabet. The appearance of all letters and digits may be seen from section 6.
7.1.1.2. Delimiters. As apparent from section 6 the following simple reference language symbols are directly available in GIER ALGOL:
+ - × / < = > ∨ ∧ , . $_{10}$: ; ( ) [ ]

7.1.2. Compound symbols.
Compound symbols must appear exactly as shown in this section, without additional SPACE or CARRET symbols.
7.1.2.1. Underlined words. Underlined words are produced in GIER ALGOL by depressing the underline (_) key immediately preceding each letter of the word. The symbols are the following:
true false go to if then else for do step until while comment begin end own Boolean integer real array switch procedure string label value
Boolean and boolean may be used interchangeably. Also go to, goto, and go to.
7.1.2.2. Compound symbols similar to reference language. The following compound symbols, most of which are produced by combining the underline (_) or stroke (|) with other characters, are similar to those of the reference language:
≤ ≥ ≠ ≡ :=
7.1.2.3. Compound symbols differing from reference language. The following compound symbols show a noticable deviation from the reference language:

Reference language	↑	¬	␣	(	)	+	⊃
GIER ALGOL	↑	-,	⊥	⟨	⟩	:	=>

7.2. USE OF comment.

Following the delimiter comment any sequence of characters specified in section 6.5 is admitted up to the first following semicolon (;). Comments have no effect in GIER ALGOL.

7.3. THE TREATMENT OF VARIABLES OF TYPES integer AND real.

Variables of types integer and real are represented by normal floating point numbers in GIER. Therefore integers must be confined to the range:

$$-2\uparrow 29 = -536\,870\,912 \leq \text{integer} \leq 536\,870\,911 = 2\uparrow 29 - 1$$

while the range of non-zero real variables is:

$$2\uparrow(-512) = 7.458_{10}{-155} < \text{abs(real)} < 1.341_{10}154 = 2\uparrow 512$$

If in the course of a calculation an expression, which according to the rules of section 3.3.4 is of type integer, yields a result outside the range for integers, the result will be represented by too few significant figures and will therefore in general be inexact.

Round-off from type real to type integer is performed by means of the built-in machine instructions for conversion from floating form to fixed form and back again (tkf -29, nkf 39). This implies that real results in the range from 0 to 2↑29 will yield correct integers on rounding, while reals in the range from 2↑29 to 2↑39 will be rounded to an integer having too few significant figures. Real results larger than 2↑39 will yield completely erroneous results if rounded.

The accuracy of a real number will correspond to 29 significant binary digits. Thus one unit in the last binary place will correspond to a relative change of the number of between 2_{10}-9 and 4_{10}-9.

7.4. RESERVED IDENTIFIERS.

A reserved identifier is one which may be used in a program for a standard purpose without having been declared in the program. If the standard meaning is not needed in a program the identifier may freely be declared to have other meanings.

The complete list of reserved identifiers arranged alphabetically is as follows:

Identifier	Reference	Identifier	Reference
abs	3.2.4	streng	9.6
arctan	3.2.4, 7.5	sættegn	9.10
cos	3.2.4, 7.5	tast	9.8
entier	3.2.5, 7.5	taststreng	9.8
exp	3.2.4, 7.5, 11.7	tasttegn	9.9
fra tromle	10.5	tegn	9.11
ln	3.2.4, 7.5, 11.7	til tromle	10.5
lyn	9.12	tromleplads	10.5
læs	9.4	tryk	8.3
læsstreng	9.6	trykende	8.7
læst	9.5	trykklar	8.7
læstegn	9.9	trykkopi	9.7
sign	3.2.4	trykml	8.5
sin	3.2.4, 7.5	trykslut	8.7
skrv	8.3	trykstop	8.6
skrvkopi	9.7	tryksum	8.7
skrvml	8.5	tryktab	8.6
skrvtab	8.6	tryktegn	8.8
skrvtegn	8.8	tryktekst	8.4
skrvtekst	8.4	tryktom	8.5
skrvvr	8.6	trykvr	8.6
sqrt	3.2.4, 7.5		

7.5. STANDARD FUNCTIONS.

7.5.1. Accuracy.

The algorithms for calculating the standard functions arctan, cos, exp, ln, sin, and sqrt, incorporated in GIER ALGOL will all yield results having an error less than that which corresponds to about 2 units in the last place of the result or the argument, whichever gives the greater error.

7.5.2. Alarms.

Certain misuses of the standard functions will cause termination of execution of program (see section 11.7). Note, however, that ln(0) will supply the result $-9.35_{10}49$ and not call the alarm.

7.6. ARITHMETIC EXPRESSIONS.

The treatment of arithmetic types and the accuracy of real arithmetics is described in section 7.3. Alarms are described in section 11.7.

7.7. (This section has been deleted).

7.8. INTEGERS AS LABELS.

Integers cannot be used with the meaning of labels in GIER ALGOL.

7.9. FOR STATEMENTS.

In GIER ALGOL a subscripted variable is permitted as the controlled variable in a for clause. The identity of the variable will be established once at the beginning of each activation of the for statement and changes of the values of subscript expressions in the course of the execution of the controlled statement will have no influence on which variable is used as the controlled one.

7.10. PROCEDURE STATEMENTS.

7.10.1. Recursive procedures.

Recursive procedures will be processed fully in GIER ALGOL.

7.10.2. Handling of types.

The types integer and real will be handled according to the prescriptions of section 4.7.3 except in the case that a formal parameter, which is specified to be real and to which assignments are made, in the call corresponds to an integer declared variable. This special case will be treated incorrectly in GIER ALGOL.

7.10.3. Extended list of standard procedures.

All input and output functions are in GIER ALGOL expressed as calls of standard procedures. These calls conform to the syntax of calls of declared procedures (cf. section 4.7.1) and also should be regarded in all other respects as regular procedure calls or function designators, as the case may be. This specifically includes the activation of a standard procedure through its identifier appearing as an actual parameter of a call of a declared procedure.

7.11. ORDER OF DECLARATIONS.

In GIER ALGOL declarations may appear in any order in the block head.

7.12. Own.

In GIER ALGOL own can only be used with type declarations, not with array declarations.

7.13. PROCEDURE DECLARATIONS.

7.13.1. Recursive procedures.

Recursive procedures will be processed fully in GIER ALGOL.

7.13.2. Arrays called by value.

GIER ALGOL cannot handle arrays called by value.

7.13.3. Specifications.

The specifications for formal parameters must be complete, i.e. each parameter must occur just once in the specification part.

7.13.4. Labels called by value.

Labels cannot be called by value in GIER ALGOL (the Revised ALGOL 60 Report leaves the question unanswered).

7.14. GENERAL LIMITATIONS.

GIER ALGOL imposes a number of limitations caused by the finite size of the tables used during compilation. However, with one exception these limitations shall not be mentioned further here, partly because only very exceptional programs are likely to exceed the capacity, partly because alarm messages during compilation will indicate when they are violated (see appendix 4). The exception is the limitation that the number of variables which are active simultaneously at any time during the execution of a program must be confined to about 700. This problem is discussed in detail in section 10.

8. STANDARD OUTPUT PROCEDURES.

Output of text and results from a program will be controlled by means of output procedures permanently available to the translator (i.e. without explicit declarations). The output will be provided in the form of 8-channel punch tape or printed copy. The symbols and 8-channel code given in section 6. 8-CHANNEL PUNCH TAPE CODE AND FLEXOWRITER KEYBOARD will be used.

8.1. CONTROL OF TYPEWRITER AND OUTPUT PUNCH.

Half of the standard output procedures are available in two forms, one controlling the output punch (identifier beginning with tryk), the other controlling the on-line typewriter (identifier beginning with skrv). By operator intervention it is however possible to make a free choice of the output unit corresponding to the two sets of output procedure identifiers. See the section 11.6 CHOICE OF OUTPUT UNITS OR STOP RUN.

8.2. IDENTIFIERS AND MAIN CHARACTERISTICS.

The identifiers and main characteristics of the standard output procedures are the following:

Identifier	Example, reference	Effect
tryk skrv	tryk({+d,ddd}, q, 2) section 8.3.	Outputs the values of an arbitrary number of arithmetic expressions in a specified layout. Other output operations may also be inserted as parameters.
tryktekst skrvtekst	skrv({<Q,=,}) section 8.4.	Outputs a specified string of symbols.

trykml skrvml	trykml(8-n) section 8.5.	Outputs a specified number of SPACEs.
tryktom	tryktom (100) section 8.5.	Punches a specified number of TAPE FEED symbols.
trykvr skrvvr	skrvvr section 8.6.	Outputs one CAR RET symbol.
tryktab skrvtab	tryktab section 8.6.	Outputs one TAB symbol.
trykstop	trykstop section 8.6.	Punches one STOP CODE symbol.
trykende	trykende section 8.7.	Punches one END CODE symbol.
trykslut	trykslut section 8.7.	Punches one PUNCH ON symbol.
trykklar	trykklar section 8.7.	Punches one CLEAR CODE symbol and sets internal sum of punched symbols to zero.
tryksum	tryksum section 8.7.	Punches a STOP CODE, a SUM CODE and a code representing the sum of the symbols punched since program read-in, last trykklar or last tryksum.
tryktegn skrvtegn	skrvtegn (49) section 8.8	Outputs the character corresponding to the value of the parameter.
trykkopi skrvkopi	trykkopi ({</;}) section 9.7.	Copies a section of the input tape to the output, the section being specified through a parameter.

It holds for all standard output procedures that each output operation will cause an addition to an internal variable of a number which is equivalent to the character. This may be used for checking purposes by means of the mechanisms described in sections 8.7.2 and 9.2. It should be noted, however, that for the checking to work correctly the output tape must not include any character which has been produced by a skrv - operation (cf. section 8.1).

8.3. STANDARD PROCEDURES: tryk, skrv.

8.3.1. Syntax.

```
<sign> ::= <empty>| - | + | +
<exponent layout> ::= b<sign>d | <exponent layout>d
<zeroes> ::= 0 |<zeroes>0 | <zeroes>,0
<positions> ::= d | <positions>d | <positions>,d
<0-positions> ::= <positions> | <0-positions>0 | <0-positions>,0
<decimal layout> ::= <0-positions>|<0-positions>.<zeroes>|
                     <positions>.<0-positions>|.<0-positions>
<layout tail> ::= <decimal layout>|<decimal layout><exponent layout>
<layout> ::= <sign><layout tail>|<sign>n<layout tail>|<sign> n|
             <sign>n,<layout tail>
<general layout> ::= {<layout>}|<formal parameter>|(<layout expression>)
<layout expression> ::= <general layout>|
    <if clause><general layout> else <layout expression>
<output statement> ::= <tryk statement>|<tryktekst statement>|
    <trykml statement>|<tryktom statement>|<trykvr statement>|
    <tryktab statement>|<trykstop statement>|<trykende statement>|
    <trykslut statement>|<trykklar statement>|<tryksum statement>|
    <trykkopi statement>|<tryktegn statement>
<tryk parameter> ::= <arithmetic expression>|<output statement>
<tryk parameter list> ::= <tryk parameter>|
    <tryk parameter list>,<tryk parameter>
<tryk statement> ::= tryk(<layout expression>,<tryk parameter list>)|
                     skrv(<layout expression>,<tryk parameter list>)
```

8.3.2. Examples.

```
tryk({ddd.00}, P, trykvr, tryktekst ({<Q=>}), w +s)
skrv ({-d10-dd}, epsilon/16)
tryk({dd⊥ddd}, Q, trykml(5), tryk({.ddd}, q), W, t-3)
tryk(if s>0 then f1 else f2, Sum)
tryk(1, p-q, s+t)
```

8.3.3. Semantics.

A call of the procedure tryk or skrv causes the following treatment of the parameters specified in the tryk parameter list:

Arithmetic expression: the value will be printed in the layout supplied in the first parameter of the call.

Output statement: the call of the statement will be executed.

8.3.4. The layout.

The layout expression will be evaluated once at the beginning of the execution of the tryk or skrv statement. The evaluation will take place in a way which is completely analogous to that of other expressions (cf. section 3.3.3). The final value must always be of the form {<layout>}.

The symbols of the layout give a symbolic representation of the digits, spaces and symbols as they will appear in the printed number. Indeed, the finally printed number will have exactly the same number of printed characters as is present in the layout (except in case of alarm printing, see section 8.3.6). The various symbols of the layout have the following significance:

8.3.4.1. Sign. The four possible symbols in the sign position signify the following:
8.3.4.1.1. Empty. The number is supposed to be positive. No sign will be printed. If a negative number is encountered, an alarm printing will take place (see section 8.3.6).
8.3.4.1.2. - . The sign will always be printed using SPACE for positive, and - for negative numbers. It will, if possible, move to the right, appearing as the first or second symbol to the left of the first digit (a layout SPACE may appear in between) or immediately in front of the decimal point.
8.3.4.1.3. + . The sign will always be printed using + for positive and - for negative numbers. It will, if possible, move to the right, as in 8.3.4.1.2 above.
8.3.4.1.4. $\pm$. The sign will always be printed, using + for positive and - for negative numbers. It will be printed as the first symbol of the number, before any SPACE or digit.

8.3.4.2. Digits. Letters d and n represent digits. Letter n may only appear as the first symbol following the sign. The total number of letters d and n gives the maximum number of printed significant digits (cf. section 8.3.8).
If n is used in the first digit position, proper decimal fractions will be printed with a 0 in front of the decimal point and the integer 0 will be printed. If d is used these 0-digits will be replaced by SPACE.

8.3.4.3. Zeroes. Zeroes may appear at the end of a decimal layout. They influence the representation of the number in the following manner: If m zeroes are present at the end of the decimal layout the exponent printed will be exactly divisible by m+1. For this to be possible at the same time as the position of the decimal point within the complete layout is kept fixed the significant digits of the number are allowed to move to the right, using the positions of the symbols 0, depending on the magnitude of the number. If no exponent layout is included the exponent 0 is understood and the above rule holds unchanged.

8.3.4.4. Spaces. Spaces will be inserted in all positions where the symbol $\perp$ appears. The symbol $\perp$ may within the layout be replaced by SPACE the effect of SPACE being the same.

8.3.4.5. Decimal point. The decimal point will always be printed in a fixed position within the layout. If decimals are printed it will appear as . otherwise as SPACE.

8.3.4.6. Scale factor. The scale factor will be printed in the same way as in the language. The symbol $_{10}$ will appear immediately in front of the sign of the exponent. If the scale factor is 1 the symbols $_{10}$ and following will appear as SPACEs. Note that it is not possible to print an exponent part without a decimal part.

8.3.5. Round-off.
All numbers will be correctly rounded to the number of significant digits printed.

8.3.6. Limitations.

The total number of symbols n and d in any decimal layout must be ≤ 15.

The total number of symbols n, d, and 0, written to the left of the decimal point must be ≤ 15.

The total number of symbols d and 0 written to the right of the decimal point in a decimal layout must be ≤ 15.

The number of symbols d in any exponent layout must be ≤ 7.

The symbols , and SPACE can only appear in such positions within the layout that they are preceded by fewer than 20 symbols of the kinds n, d, 0, and point (.).

8.3.7. Alarm printing.

By alarm printing is meant that the printing will consume more positions on the paper than are present in the layout. Alarm printing will occur as follows:

8.3.7.1. Negative number printed with layout having empty sign position. The correct - will be inserted, consuming one extra position.

8.3.7.2. Number too large for layout. Whenever the number to be printed is too large for the layout given, an actual layout is used which will accomodate the number by inserting an exponent layout, or by increasing the number of exponent digits.

8.3.8. Small numbers.

Printing of small numbers will never give rise to alarm printing. Instead the number of printed significant digits will be smaller than the maximum (section 8.3.4.2).

8.3.9. Examples of printed numbers.

In order to indicate the exact number of characters printed, commas are inserted immediately preceding and following each number.

Layout n␣dd␣dd.d0␣0	+d␣ddd.ddd␣d	-ddd.d00$_{10}$+d	±dd.0$_{10}$-dd
Normal printing			
, 0.00 1,	, +.001 2,	, 1.235$_{10}$-3,	,+12 $_{10}$-4,
, 0.01 2,	, +.012 3,	, 12.35 $_{10}$-3,	,+ 1.2 $_{10}$-2,
, 0.12 3,	, +.123 5,	, 123.5 $_{10}$-3,	,+12 $_{10}$-2,
, 1.23 5,	, +1.234 6,	, 1.235 ,	,+ 1.2 ,
, 12.34 6,	, +12.345 7,	, 12.35 ,	,+12 ,
, 1 23.45 7,	, + 123.456 8,	, 123.5 ,	,+ 1.2 $_{10}$ 2,
, 12 34.57 ,	,+1 234.567 9,	, 1.235$_{10}$+3,	,+12 $_{10}$ 2,
,1 23 45.7 ,		, 12.35 $_{10}$+3,	,+ 1.2 $_{10}$ 4,
	, -.001 2,	, -1.235$_{10}$-3,	,-12 $_{10}$-4,
	,-1 234.567 9,	, -1.235$_{10}$+3,	,-12 $_{10}$ 2,
Alarm printing			
, -0.00 1,			
,-1 23 45.7 $_{10}$3,	,-1 234.567 9$_{10}$4,		
, 1 23.45 7$_{10}$15,	,+1 234.567 9$_{10}$14,	, 123.5 $_{10}$+15,	

8.4. STANDARD PROCEDURES: tryktekst, skrvtekst.

8.4.1. Syntax.

<general string> ::= {<<proper string>} |<formal parameter>|
(<string expression>)
<string expression> ::= <general string>|
<if clause><general string> else <string expression>
<tryktekst parameter> ::= <string expression>|<output statement>
<tryktekst parameter list> ::= <tryktekst parameter>|
<tryktekst parameter list>,<tryktekst parameter>
<tryktekst statement> ::= tryktekst(<tryktekst parameter list>)|
skrvtekst(<skrvtekst parameter list>)

8.4.2. Examples.

tryktekst({<<Result is}, a, {<<than expected})
skrvtekst({<<Q␣=␣})

8.4.3. Semantics.

The execution of a tryktekst statement causes the following treatment of the parameters specified in the parameter list, taking them in order from left to right:

String expression: an output of the text resulting from an evaluation of the expression is performed.

Output statement: the call of the statement will be executed.

8.4.3.1. The string quote.

Note the difference between the string quotes used here

{< }

and those used in layout expressions (cf. section 8.3.1).

8.4.3.2. Treatement of SPACE and CAR RET.

All characters of the proper string, including SPACEs and CAR RETs will be outputed. The symbol for space ␣ will however be equivalent to SPACE, i.e. it will be printed, not as it stands, but as a SPACE.

8.5. STANDARD PROCEDURES: trykml, skrvml, tryktom.

8.5.1. Syntax.

<trykml statement>::= trykml (<arithmetic expression>)|
skrvml (<arithmetic expression>)
<tryktom statement>::= tryktom (<arithmetic expression>)

8.5.2. Examples.

trykml(n + m - 7)
tryktom (75)
skrvml (if p > 0 then 3 else 4)

8.5.3. Semantics.

The execution of a trykml statement causes the number of SPACE symbols (mellemrum) specified as actual parameter to be outputed.

A call of the procedure tryktom causes the number of TAPE FEED symbols specified as actual parameter to be outputed.

The value of the arithmetic expression will, if necessary, be rounded to the nearest integer. If it assumes a non - positive value no symbols will be outputed.

8.6. STANDARD PROCEDURES: trykvr, skrvvr, tryktab, skrvtab, trykstop.

8.6.1. Syntax.

<trykvr statement>::= trykvr|skrvvr
<tryktab statement>::= tryktab|skrvtab
<trykstop statement>::= trykstop

8.6.2. Semantics.

A trykvr statement causes a CAR RET symbol (vogn retur) to be outputed. Note that this will cause the combined operation of return of carriage and line feed to take place.

A tryktab statement causes output of a TAB symbol.

A trykstop statement causes the STOP CODE to be punched.

8.7.1. Syntax.

<trykende statement>::= trykende
<trykslut statement>::= trykslut
<trykklar statement>::= trykklar
<tryksum statement>::= tryksum

8.7.2. Semantics.

The four output procedures described here all serve to insert characters on the output tape with a view to a later use of this output tape as input tape to an ALGOL program.

The trykende statement punches the END CODE. When later the tape is read into the machine this will cause a stop of the machine (cf. section 9.2.6).

The trykslut statement punches the PUNCH ON symbol. This is intended to be used as a non - printing terminator for last and last (cf. sections 9.4 and 9.5).

The trykklar statement punches the CLEAR CODE and sets the internal sum of the punched characters to zero. This prepares for the use of the checksum mechanism (cf. section 9.2.5).

The tryksum statement punches a STOP CODE, a SUM CODE and a character representing the value of the internal sum of all punched characters and sets this sum to zero. During input this combination will cause an automatic sum check to take place (cf. section 9.2.5).

8.8. STANDARD PROCEDURES: tryktegn, skrvtegn.

8.8.1. Syntax.

<tryktegn statement> ::= tryktegn(<arithmetic expression>) |
skrvtegn(<arithmetic expression>)

8.8.2. Examples.

tryktegn(if upper case then 60 else 58)
skrvtegn(49)
skrvtegn(symbol - case)

8.8.3. Semantics.

The execution of a tryktegn statement causes the character corresponding to the value of the actual parameter to be outputted. The correspondence between the integers and the characters is given in the table of section 6.5. If the value of the actual parameter is not an integer it will be rounded to the nearest integer. If it is larger than 127 the value modulo 128 will be used.

The characters for UPPER CASE and LOWER CASE must be outputted explicitly where needed. Where tryktegn statements are used side by side with tryk or tryktekst statements it is important to note that these latter will assume the output unit to be in lower case when a call is made and will also leave it in lower case when the call is completed.

9. STANDARD INPUT PROCEDURES.

Input of information from 8-channel punch tape may be carried out at any stage of an ALGOL program through calls of standard input procedures permanently available to the translator.

In order to provide flexibility several different kinds of standard input procedures are available. These differ both with respect to the interpretation of the single symbols supplied on the input tape and the internal effect of the input operation.

9.1. IDENTIFIERS AND MAIN CHARACTERISTICS.

The identifiers and main characteristics of the standard input procedures and the associated procedure streng are the following:

Identifier	Example, reference	Effect
læs	læs(a, b, c) section 9.4.	Reads numbers and assigns to variables or arrays.
læst tast	p × læst section 9.5, 9.8.	real procedures læst and tast have the next number appearing on the input tape or typed on the typewriter as their value.
læsstreng taststreng	læsstreng section 9.6, 9.8.	Read a string of symbols from tape or typewriter to an internal variable for later comparison by means of the
streng	streng(‡<P‡) section 9.6.	boolean procedure streng. The value of streng is true if the string supplied as parameter agrees with the string read by the last call of læsstreng.
trykkopi skrvkopi	trykkopi(‡</;‡) section 9.7.	Cause a copying of the characters on the input tape to the output punch (trykkopi) or the typewriter (skrvkopi).
læstegn tasttegn	n:= tasttegn section 9.9.	These integer procedures supply the value of the next character which appears on the tape or is typed.
sættegn	sættegn (15) section 9.10.	Inserts an input character ahead of the ones waiting in the input.
tegn	p:= tegn section 9.11.	Supplies the value of the last character read by any input procedure.
lyn	q:= lyn + 4 section 9.12.	Supplies the value of the next row of holes on the input tape.

9.2. UNIVERSAL INPUT MECHANISMS.

Certain characters on the input tape will be handled in the same way no matter which of the standard input procedures is controlling the input operation. The universal mechanisms are the following:

9.2.1. Skipping between PUNCH OFF and PUNCH ON.

All characters between PUNCH OFF and the first following PUNCH ON, these two characters included, will be completely ignored during input.

9.2.2. Ignoring of BLANK TAPE, TAPE FEED, and ALL HOLES.

The characters

.	BLANK TAPE
oooo.ooo	TAPE FEED
ooooo.ooo	ALL HOLES

will be ignored during input.

9.2.3. (This section has been deleted).

9.2.4. Input characters of wrong parity.

The machine stops when a row of an even number of holes is sensed in the tape reader. In this situation it is sufficient to place the intended symbol in the R register since the ALGOL system never makes any use of the representation stored by the input instruction itself.

9.2.5. The checksum mechanism.

When the standard input procedures read tapes which have been prepared by the standard output procedures the checksums included on this tape in consequence of calls of the tryksum procedure will automatically be verified. If the check symbol does not check with the corresponding symbol as formed during previous read-in the machine will print

sumfejl

and the machine will stop. If a character is typed on the typewriter the reading will continue. The internal variable which holds the current sum of the symbols which have been read in may be reset to zero by the inclusion of the CLEAR CODE on the tape. This is the symbol produced by the trykklar procedure (cf. section 8.7.2). On the flexowriter use:

AUX CODE with 0

9.2.6. Stop produced by END CODE.

Whenever the END CODE appears the message

vent

will be typed and the machine will stop, waiting for a character to be typed on the typewriter. The END CODE may be produced by an ALGOL program by a call of the trykende procedure (cf. section 8.7.2). On the flexowriter it is produced by depressing

AUX CODE with SPACE.

9.2.7. The effect of UPPER CASE and LOWER CASE.

For printed symbols (cf. section 6.1) the meaning and effect of a given hole combination depends on the most recent CASE symbol on the tape (UPPER CASE or LOWER CASE).

For typographical and control symbols (cf. sections 6.2 and 6.3) the effect is usually independent of the case.

9.3. TERMINATORS, INFORMATION SYMBOLS, AND BLIND SYMBOLS.

The effect of the input characters which do not give rise to an action of a universal input mechanism (cf. section 9.2) depends on the particular standard input procedure. In describing this effect it is convenient to make use of the following concepts:

9.3.1. Terminators. A terminator is a symbol on the input tape which indicates to the input procedure that the reading of a piece of information (e.g. a number) has been completed.

9.3.2. Information symbols. An information symbol is a symbol on the input tape supplying positive information which is transferred to the running ALGOL program by the input procedure.

9.3.3. Blind symbols. A blind symbol is a symbol on the input tape which is ignored by the input procedure.

As explained more concisely in the following sections we have for the procedures læs and læst:

Terminators: <letter> all signs except $+-.{}_{10}$ TAB PUNCH ON CAR RET
Information symbols: <digit> $+-.{}_{10}$
Blind symbols: SPACE _ STOP CODE
and for læsstreng:
Terminators: all signs TAB PUNCH ON CAR RET
Information symbols: <digit> <letter>
Blind symbol: SPACE _ STOP CODE

Each input operation will in general read three sections of the input tape:

1. Any mixture of terminators and blind symbols.
2. A legal sequence of information symbols mixed with blind symbols.
3. One terminator.

9.4. STANDARD PROCEDURE: læs.

9.4.1. Syntax.

<læs parameter>::= <variable>|<array identifier>
<læs parameter list>::= <læs parameter>|
<læs parameter list>,<læs parameter>
<læs statement>::= læs(<læs parameter list>)

9.4.2. Examples.

```
læs(P)
læs(A[i,j], V, MATA)
læs(k, B[1,k])
```

9.4.3. Semantics.

A call of the procedure læs will cause the values of numbers supplied on the input tape to be assigned to the variables and/or arrays of subscripted variables specified as parameters. The assignments will in detail be executed as follows:

9.4.3.1. Order of assignment. The parameters will be taken in order from left to right and the assignment will be completely finished for each parameter before the next is treated. Thus the statement læs(k, B[1,k]) will first assign a value from the input tape to k and this value of k will then define the particular component of B to which the next number on the tape will be assigned.

9.4.3.2. Assignment to array. If an array identifier is supplied as parameter an assignment to all the components of the array will take place. The order of assignment may be described as follows: Denoting the lower and upper subscript bounds of the array declaration by l1, l2, ... ln, u1 u2, ... un, the input operation is equivalent to

```
for i1:= l1 step 1 until u1 do
for i2:= l2 step 1 until u2 do
     .......

for in:= ln step 1 until un do
 A[i1, i2, ... , in]:= input number
```

where i1, i2, ... in are internal variables.

9.4.3.3. Input tape syntax. The characters appearing on the input tape during the execution of læs must conform to the following syntactic rules:

```
<læs terminator>::= ∨|×|/|=|;|[|]|(|)|||∧|<|>|,|TAB|PUNCH ON|:|CAR RET|
                    <letter>
<læs information>::= <digit>|.|10|+|-
<læs blind>::= SPACE|_|STOP CODE
<læs prelude>::= <empty>|<læs blind>|<læs terminator>|
      <læs prelude><læs blind>|<læs prelude><læs terminator>
<digit sequence>::= <digit>|<digit sequence><digit>|
      <digit sequence><læs blind>|<læs blind><digit sequence>
<input integer>::= <digit sequence>|+<digit sequence>|-<digit sequence>
<input fraction>::= .<digit sequence>
<input exponent>::= 10<input integer>
<input decimal>::= <digit sequence>|<input fraction>|
                   <digit sequence><input fraction>
<unsigned real>::= <input decimal>|<input exponent>|
                   <input decimal><input exponent>
<input real>::= <unsigned real>|+<unsigned real>|-<unsigned real>
<input ditto>::= -|<input ditto>-|<input ditto><læs blind>
<tape integer>::= <læs prelude><input integer><læs terminator>|
                   <læs prelude><input ditto><læs terminator>
<tape real>::= <læs prelude><input real><læs terminator>|
               <læs prelude><input ditto><læs terminator>
```

9.4.3.4. Examples of input tape for læs.

```
Tape integers:
17 283;
i = +138,
S[25]
funktion(-12)
p: -/
```

```
Tape reals:
w:= 3.857 392 <
eps:= -10-14,
pi:= 3.141592 65;
Sæt x = 4,
q: 1.38410-11,
```

9.4.3.5. Semantics of input tape. Depending on the type of the variable each læs assignment will cause the reading of one tape real or tape integer. If these contain digits they will be interpreted according to the usual ALGOL prescriptions (cf. sections 2.5.3 and 2.5.4), ignoring all læs blinds and læs terminators. An input ditto, on the other hand, will cause the læs assignment to be skipped for the particular variable, thus leaving its value unchanged.

9.4.3.6. Errors. The standard procedure læs checks that the syntactic rules of section 9.4.3.3 are satisfied. If an error is detected one of the messages

talfejl. tast her, slut i LC:

or

talfejl. tast her, slut i UC:

will be typed. The operator is now expected to type one number, followed by a terminator, to be used instead of the erroneous combination appearing on the tape. The terminator must be in upper or lower case as indicated in the message since otherwise the following text on the input tape may be misinterpreted.

9.5. STANDARD PROCEDURE: læst.

9.5.1. Syntax.
<læs function designator>::= læst

9.5.2. Examples.
w:= (læst + y)/q
B[læst, læst]:= læst

9.5.3. Semantics.
læst is a real procedure having an empty formal parameter part. Every time it is called it will read the next tape real appearing on the input tape (cf. section 9.4.3.3). This information on the input tape will define its value according to the rules of section 9.4.3.5, except that the effect of an input ditto is undefined.

9.5.3.1. Example of input tape for læst. A reasonable input tape for the second example of section 9.5.2 would be the following:
B[3,7]:= 3.847,
Note that the correct execution of this input operation is directly dependent on the strict adherence to the rules of sections 4.2.3.1 - 4.2.3.3 for assignment statements.

9.6. STANDARD PROCEDURES: læsstreng, streng.

9.6.1. Syntax.

<læs streng statement>::= læsstreng
<streng function designator>::= streng(<string expression>)

9.6.2. Examples.

læs streng
if streng(A) then go to T

9.6.3. Semantics.

The standard procedures læsstreng and streng serve to read identifying information from the input tape and to compare this information with information supplied by the program. The detailed operation is defined below.

9.6.3.1. Input tape syntax. During execution of læsstreng the characters on the input tape are treated according to the following syntax:

<læsstreng terminator>::= ∨|×|/|=|;|[|]|(|)|||∧|<|>|,|₁₀|TAB|-|+|PUNCH ON|.|:|CAR RET
<læsstreng information>::= <digit>|<letter>
<læsstreng blind>::= SPACE|_|STOP CODE
<læsstreng prelude>::= <empty>|<læs streng blind>|
    <læsstreng terminator>|<læsstreng prelude><læsstreng blind>|
    <læsstreng prelude><læsstreng terminator>
<input string>::= <læsstreng information>|<input string><læsstreng blind>|
    <input string><læsstreng information>
<tape string>::= <læsstreng prelude><input string><læsstreng terminator>

9.6.3.2. The internal string. Each call of læsstreng will read the first following tape string from the input tape and assign the five first information symbols of the input string, which is a part of it, to a unique internal variable. If the input string has less than five information symbols it will be extended with the appropriate number of unique dummy characters.

9.6.3.3. Examples of tape strings and internal strings.

Symbols on tape	Internal string
b7.	b7
(Matrix A)	Matri
[×]:A and B;	AandB
true,	true

9.6.3.4. Standard procedure streng. This is a boolean procedure, requiring a string expression as parameter. It has the value true if all the characters of the value of the string expression agree with the same number of characters of the internal string, assigned by the previous læsstreng, both strings taken in order from left to right, otherwise the value false. Note that the agreement of the two strings puts the following restrictions on the string supplied as parameter to streng:
9.6.3.4.1. It cannot contain more characters than the number of information symbols in the internal string (never more than 5).
9.6.3.4.2. It can only contain digits and letters.

9.6.3.5. Example. The following table shows the value of streng for various input strings and parameters:

	Parameter:		
Input string	A	Alg	ALGOL
ALGOL 60	true	false	true
A	true	false	false
Blg	false	false	false
Algol	true	true	false
Algorithm	true	true	false

9.7. STANDARD PROCEDURES: trykkopi, skrvkopi.

9.7.1. Syntax.
<trykkopi statement>::= trykkopi(<string expression>)|
skrvkopi(<string expression>)

9.7.2. Examples.
trykkopi(‹<+/›)
skrvkopi(if s>0 then w else y)
trykkopi(fs)

9.7.3. Semantics.
A call of a trykkopi statement causes a copying of characters from the input tape to the output. The section of the input tape to be copied is defined by the value of the string expression supplied as parameter. This value must have the form

‹< <proper string> ›

where the proper string consists of one or two characters. If one character is supplied the copying will take place from the actual position of the input tape until the first occurrence of the character specified as parameter. If two characters are supplied the copying will start from the first character on the tape which is the same as the first of the two characters supplied as parameters and will continue until the first occurrence of the second of these symbols on the tape. The characters indicating the begin and end of the section of the input tape to be copied will not themselves be copied.
The copying will include all legal characters except those associa-

ted with the universal input mechanisms (cf. section 9.2) and superfluous case shifts.

9.7.3.1. Example of call, input tape, and output.
The call
trykkopi(‡<[]‡)
operating on the following input tape:
Heading: [
Problem number:]
will produce as output:
Problem number:

9.8. STANDARD PROCEDURES: tast, taststreng.

These procedures are entirely similar to procedures læst and læsstreng (sections 9.5 and 9.6) but expect the input characters to be typed on the typewriter.

9.9. STANDARD PROCEDURES: læstegn, tasttegn.

9.9.1. Syntax.
<læstegn function designator> ::= læstegn | tasttegn

9.9.2. Examples.
if tasttegn = 49 then go to a
symbol := læstegn

9.9.3. Semantics.
læstegn and tasttegn are integer procedures having an empty formal parameter part. Each call of a læstegn function designator will activate the corresponding input unit (paper tape reader for læstegn, typewriter for tasttegn) and will return with the value of the next proper character from the input medium as its value. By proper character is here meant a character which is not handled by the universal input mechanisms (section 9.2). The values of proper characters in lower case are given directly by the table in section 6.5. In upper case the value supplied by læstegn and tasttegn is increased by 128. Thus the letter p will appear as 39 while P will be 167.

9.10. STANDARD PROCEDURE: sættegn.

9.10.1. Syntax.
<sættegn statement> ::= sættegn(<arithmetic expression>)

9.10.2. Examples.
sættegn(160)
sættegn(tegn)

9.10. STANDARD PROCEDURE: sættegn.

9.10.3. Semantics.

Each call of sættegn assigns the value of the expression supplied as actual parameter to an internal buffer and at the same time sets an internal Boolean variable which causes the value in the buffer to be used as the first proper input character at the first following call of any input procedure (læs, læst, tast, læsstreng, taststreng, læstegn, tasttegn, trykkopi, skrvkopi) ahead of the next symbol waiting in the input unit.

The values of the actual parameters supplied in calls of sættegn should only be such which correspond to proper input characters, i.e. such which may appear as values of læstegn.

9.11. STANDARD PROCEDURE: tegn.

9.11.1. Syntax.

<tegn function designator> ::= tegn

9.11.2. Examples.

```
if tegn < 10 then tryktegn(tegn)
if tegn = 133 then go to exit
```

9.11.3. Semantics.

tegn is an integer procedure having an empty formal parameter part. Its value is the number corresponding to the last proper character previously inputted by any standard input procedure (læs, læst, tast, læsstreng, taststreng, læstegn, tasttegn, trykkopi, skrvkopi) or assigned by sættegn. The value corresponding to a proper character is to be understood in the same sense as for procedure læstegn. Note that tegn does not activate any input unit, but only makes the last character supplied by any input unit available.

9.12. STANDARD PROCEDURE: lyn.

9.12.1. Syntax.

<lyn function designator> ::= lyn

9.12.2. Example.

```
symbol:= lyn
```

9.12.3. Semantics.

lyn is an integer procedure having an empty parameter part supplying the value of a character from the paper tape reader, like læstegn. However, the character whose value is provided by lyn is always the next one on the input tape without any intervention from the universal input mechanisms (section 9.2) or the buffer controlled by sættegn (section 9.10). Likewise the case and buffer state are unaffected by calls of lyn. Thus by using lyn the programmer may interpret the input symbols having correct parity in any conceivable manner.

10. STORING VARIABLES ON DRUM.

10.1. INTRODUCTION.

ALGOL programs operating with up to about 700 variables simultaneously may be handled directly by the GIER ALGOL system. However, if programs declaring more than this number of variables simultaneously are run in the system the run will be terminated before the final end has been reached (cf. section 11.7, ak and array). What has happened is that the capacity of the directly available internal store of the machine, the so-called core store, has been exceeded.

This does not mean that problems involving a larger number of variables are outside the reach of the system since there is available in the machine a storage capacity on the so-called magnetic drum of more than 12 times that of the core store. What it does mean, however, is that the user must include in his program calls of the standard procedures til tromle (Danish for: to drum) and fra tromle (Danish for: from drum) which serve to transfer variables from the core store to the drum store and back again. From the point of view of the user the magnetic drum may in this context be regarded as a new kind of input-output medium, analogous to paper tape. The two standard procedures til tromle and fra tromle are then analogous to the standard procedures tryk and læs.

However, the use of til tromle and fra tromle should not be confined to the cases where it is indispensable. In fact, execution speed considerations will often make it desirable to keep the number of active variables in the program considerably lower than the admissible upper limit.

An intelligent assessment of the factors involved requires some knowledge of the storage allocation system incorporated in GIER ALGOL. This system is therefore explained in the following sections.

10.2. STORAGE OF VARIABLES.

The reservation of core storage space for a variable is made at the time of entry into the block in the head of which the variable is declared. Similarly reservations for a block are cancelled at the time of the corresponding exit from the block. For this reason the space reserved for the variables will usually change from time to time during the execution of a program, being at every moment equal to the sum of the reservations made by those blocks and procedure bodies which are active.

The reservations made at a block entry include other quantities besides variables. The total requirements may be derived from the declarations (including the implicit ones for local labels) of the block as follows:

	Number of locations required
Simple variables, local labels, local procedures, formal parameter	One for each quantity
Array segment	Number of array identifiers + 1 + number of subscripts + total number of variables.
Switch declaration	1 + number of switch elements
Working locations	Depends on structure of program, usually only a few.
Block, procedure body	2 if normal block, 3 if procedure, 4 if type procedure.

10.3. STORAGE OF PROGRAM.

GIER ALGOL incorporates a fully automatic system for handling the transfers of program drum tracks to the core store during the execution of the program. This system will at all times attempt to make the best use of that part of the core store which is not currently reserved for variables. This section of the core store will be divided into program track places, each of 41 locations. The available places will be used for those program tracks which are required as the program execution develops. Whenever the program execution calls for a transfer to another track it is investigated whether the track is available in the core store. If it is not it is transferred to that track place which for the longest time has been left unused.

10.4. BALANCING THE USE OF THE CORE STORE.

The transfer of a drum track to the core store requires 20 milliseconds. In contrast the transfer of control to a track which is already present in the core store takes between 0.7 and 1.6 milliseconds. It is therefore clear that A PROGRAM HAVING A LARGER PART OF THE AVAILABLE CORE STORE RESERVED BY VARIABLES WILL SPEND A LONGER TIME ON TRANSFERS OF PROGRAM TRACKS TO THE CORE STORE. The importance of this loss of speed for a given number of program track places depends very strongly on the loop structure of the program. It is small if most of the execution time of the program is spent in a loop which may be held completely in the available program track places.

To assist in estimating the number of program tracks involved in a

loop which includes calls of standard procedures the arrangement of standard procedures on the tracks reserved for them is given below.

Standard procedure track	Used by
0	skrv, tryk
1	skrv, tryk
2	skrv, tryk
3	skrv, tryk, tryktom, trykml, skrvml, skrvvr, skrvtab
4	↑ with integer exponent, abs, entier, sign, sættegn, tegn, trykvr, tryktab, trykstop, trykslut, trykende, lyn
5	streng, trykklar, tryksum
6	tryktekst, skrvtekst
7	til tromle, fra tromle
8	trykkopi, skrvkopi
9	læsstreng, taststreng, sqrt
10	læs, tast, læst, tasttegn, læstegn
11	læs, tast, læst
12	trykkopi, skrvkopi, læsstreng, taststreng, læs, tast, læst, tasttegn, læstegn
13	(alarms of input, special storage)
14	cos, sin
15	arctan
16	ln, skrvtegn
17	exp, tryktegn
18	til tromle, fra tromle (tromle data A only)

These considerations indicate that in programs where the execution speed is of any concern the number of active variables in the program should be kept rather lower than the strict upper limit; a practical limit might be 500 variables. This may be achieved by using the drum as an additional store for variables.

The increase of execution speed gained by using the drum for storage of variables will be counteracted by the loss of time incurred each time these variables are transferred to or from the drum by til tromle or fra tromle. This latter transfer time is usually of the order of 1 - 2 milliseconds per variable per transfer. Whether these transfer times are of overall significance depends on the time necessary for other processing of the variables. An estimate of such processing times may be formed on the basis of the figures given in appendix 3. It will be found that the time of even a quite moderate amount of processing will overshadow the average drum transfer time.

10.5. STANDARD PROCEDURES: til tromle, fra tromle. STANDARD VARIABLE: tromleplads.

10.5.1. Syntax.

<drum transfer function designator> ::= til tromle(<array identifier>) |
fra tromle(<array identifier>)
<tromleplads variable identifier> ::= tromleplads

10.5.2. Examples.

```
Bplads := tromleplads
Bshift := til tromle(B)
tromleplads := tromleplads - Bshift
fra tromle(B)
```

10.5.3. Semantics.

The standard integer procedures til tromle and fra tromle and the associated standard integer variable tromleplads administer the handling of transfers of arrays of values to and from the drum memory of GIER. The procedure til tromle will transfer the array of subscripted variables identified in the actual parameter to the drum and acts like an assignment of values to the drum and likewise the procedure fra tromle will assign values previously transferred to the drum to the array identified in the actual parameter. In either case the part of the drum involved in the transfer is defined by the value of the integer variable tromleplads which enters into til tromle and fra tromle as a non-local identifier. Thus in order to retrieve a set of values previously transferred to the drum the procedure fra tromle must be called with tromleplads having the same value as when the corresponding call of til tromle was made. The same holds if it is desired to assign new values to a previously used section of the drum. In any case the array supplied as parameter in the drum transfer function designator must be of the same type and have the same number of subscripted variables as the one used in the corresponding call of til tromle. However, the two arrays need not have the same number of subscripts or the same subscript bounds. If the arrays differ in these respects the correspondence of elements is established by ordering the elements of each array in the same manner as they would be if they were read from tape by means of the standard procedure læs (cf. section 9.4.3.2).

Clearly the standard variable tromleplads is the key to administering values stored on the drum. In addition the programmer may use the values of the drum transfer function designators. These are closely related to tromleplads as apparent from the following 3 rules which define the behaviour of the value of tromleplads:

1. tromleplads is initialized by the compiler to a value which is the one extreme of its permissible range of variation.

2. Every call of til tromle and fra tromle will, as a side-effect, change the value of tromleplads in a direction away from the initial value supplied by the compiler towards the other extreme of its permissible range and by such an amount that the new value is the correct one to use in transferring values to the next adjecent section of the drum.

3. The amount by which tromleplads is changed through a call of til

tromle or fra tromle will be the same whenever arrays of the same type and having the same number of subscripted variables are transferred. The amount by which tromleplads is changed is available as the value of the drum transfer function designator. In other words:

new value of tromleplads = old value + til tromle(A)
new value of tromleplads = old value + fra tromle(A).

However, nothing further about the dependence of the change of tromleplads on the size and type of the array is defined generally (the precise meaning of tromleplads will change from one edition of the compiler to another).

It will be understood from these rules that as long as no explicit assignment is made to tromleplads only calls of til tromle will be in order and each of these will use a new section of the drum adjecent to the one used in the last previous call of til tromle. Before any call of fra tromle is made the programmer must make an explicit assignment to tromleplads. The values assigned to tromleplads can only be derived from its previous values possibly modified by integral multiples of the amount by which is has changed.

The programmer has his full freedom to overwrite sections of the drum which have previously been used as long as he makes sure to use only values of tromleplads which lie within the range defined by its initial value and another extreme which marks the other end of the free section of the drum. If tromleplads steps outside this range an error reaction will occur at run time and the message (cf. section 11.7)

tromle ak

will be typed. The criterion for a set of values previously transferred by til tromle to be still intact on the drum may be formulated as follows: Each section used on the drum by til tromle will be defined by an interval of the values of tromleplads, namely that defined by the value of tromleplads just before til tromle was called and its value just after the call was completed. The values transferred will still be intact as long as no call of til tromle with an overlapping interval of tromleplads has been performed.

10.5.4. Tromledata A.

The version of til tromle and fra tromle included in the compiler tape will pack the values tightly on the drum (tromledata A). When this version is used the capacity of the drum is 10 200 values reduced by 40 times the number of drum tracks used by the translated program (this latter may be derived from the pass information, see appendix 1). The last 4 440 of these values will be placed on a part of the drum which holds the compiler. Only if these are left unused will it be possible to compile programs without loading the compiler into the machine anew.

10.5.5. Tromledata B.

By a simple correction process an alternative version of til tromle and fra tromle may be included in the compiler (cf. section 11.1). This alternative version (tromledata B) will in each call use a full number of drum tracks of 40 words each. Consequently arrays of from 1 to 40 variables will require 1 drum track, arrays of from 41 to 80 values will require 2 drum tracks, etc. The capacity of the part of the drum available for program and variables is 255 tracks. Of these 111 tracks are also used by the compiler. If these are used for data further compilation will require a reloading of the compiler.

11. OPERATING THE COMPILER.

11.1. LOADING OF COMPILER INTO GIER.

The compiler tape consists of a short special input program written in SLIP language, followed by the compiler proper in binary form (HJÆLP language). The compiler will be read into the machine by SLIP (start by typing 1). If the SLIP version which stops on transfer to the program is used the machine will stop after input of the input program and will have to be restarted by pushing START.

The input is checked by summation. If the check fails the message

SUMFEJL

(in red) will be typed. In this case a new loading of the compiler will have to be attempted.

A succesful loading of the compiler takes about 75 seconds and is completed when the message

oversætter klar

is typed. The further action to be taken is described in the section on the OVERSÆTTER-KLAR-SITUATION below.

A version of the compiler which includes Tromledata B (cf. section 10.5.5) may be produced by reading a short correction tape on top of the compiler.

The compiler occupies the tracks 39-175. It places the compiled program in the tracks 319, 318, ... as far as necessary. During program execution the compiler tracks 39-64 are used in addition to the compiled program.

11.2. MANUAL JUMP TO COMPILER.

The OVERSÆTTER-KLAR-SITUATION may be called at any time during translation of ALGOL programs by transferring control to instruction 1 in the core store.

If the core store has been used for other purposes, but the compiler is known to be intact on the drum the OVERSÆTTER-KLAR-SITUATION is called by transferring track 70 to 960 and transferring control to it. On machines equipped with the optional HJÆLP-button the same effect will follow if this button is pressed and the control words

halgol
e

are typed

11.3. OVERSÆTTER-KLAR-SITUATION.

The compiler is ready to accept ALGOL programs whenever the message

oversætter klar

has been typed. In this situation the machine is waiting for symbols to be typed on the control typewriter. This leaves certain operational choices to the operator, as described in the following.

11.3.1. Start compiling.

Typing of a SPACE (or any character other than p, s, t, o, l, or i) will start the compiler translating the program with output and other compiling features defined by the other characters typed previously. If SPACE is typed immediately following the oversætter-klar-message and also KA and KB are 0 the compiler will produce no typed or punched output, input will be taken from the paper tape reader, and program sections between PUNCH OFF and the first following PUNCH ON will be ignored. Thus programs will be compiled at the highest possible speed. The compiler produces about 30 final machine instructions per second, except in the case of very short programs where the basic time of 4 seconds becomes prominent. Other compiling modes may be specified by typing any sequence of the letters p, s, t, o, l, and i, prior to the final SPACE, and by setting KA and KB at this or a later time, as described below.

11.3.2. Compilation output.

Typing of p and s selects the output unit operating during compilation, p standing for punch (perforator) and s for typewriter (skrivemaskine). If both p and s are typed the output will appear on both punch and typewriter. Whenever an output unit is specified the normal compiler output is always produced. This includes:

11.3.3. Prelude to program:

All characters on the input tape up to and including the first appearance of be (assumed to form the first characters of the first begin of the program) and the following gin are copied to the output.

11.3.4. Epilogue of program:

All characters on the input tape following the final end up to and including the first following ; (semicolon) are copied to the output.

Additional compilation output may be specified as follows (note that this presupposes a choice of output unit by typing of p or s):

11.3.5. Line output.

Typing of l causes every 10th line of the source ALGOL program to be copied to the output with its line number attached.

11.3.6. Pass information.

Typing of i causes output of the so-called pass information. This is described in appendix 1.

11.3.7. Pass output.

If KB is set to L the intermediate output from passes 1, 2, 3, 4, 5, 6, 7, and 9 will be output. The form of this output is described in appendix 2. KB may be changed at any time during compilation and pass output will be produced accordingly.

11.3.8. Program between PUNCH OFF and PUNCH ON.

If o is typed the text between PUNCH OFF and PUNCH ON is included in the program.

11.3.9. Input from typewriter.

If t is typed the compiler takes its input from the typewriter.

Input from typewriter may also be called following the vent-message (section 11.4.1).

When input is taken from the typewriter a line of text will be processed at a time and the user has the possibility of deleting the line which is being typed. Also shift to input from tape may be specified. This is achieved as follows:

11.3.9.1. A line which is terminated with the CAR RET character will be included in the program.

11.3.9.2. Whenever 4 consecutive case shifts are typed (i.e. LC, UC, LC, UC or UC, LC, UC, LC) the compiler types the message

s

(in red). If now the operator types l the compiler will complete the red message to read

slæs

(læs is Danish for: read) and the compiler will continue to take its input from tape, including the line which has just been typed. If the operator types r the compiler will complete the red message as follows:

sret

(ret is Danish for: correct) and be ready for another line to by typed instead of the previous one, which will be ignored.

11.3.10. Stop between translation passes.

If KA is set to L the machine will stop after each of the passes 1 - 8. The compiler is restarted by typing any character on the typewriter.

11.4. TYPED MESSAGES FROM COMPILER.

Irrespective of the choice of output from the compiler certain messages will be typed on the typewriter. These are

11.4.1. Vent message.

The message:

vent

(Danish for: wait) is typed and the machine stops when the END CODE is encountered on the input tape during pass 1.

If in this situation the letter t is typed the further input will be taken from the typewriter (cf. section 11.3.9). Any other character will restart the input from tape. Note that the last case shift character read from the tape will be restored correctly after shift to input from typewriter and return to input from tape.

11.4.2. Off and on messages.

Whenever the text between a PUNCH OFF and the first following PUNCH ON is ignored these two control symbols produce messages during pass 1 as follows:

linie <line number> off

and

linie <line number> on.

11.4.3. klar-message.

The message

klar

(Danish for: ready) indicates that the system is in the KLAR-SITUATION with the program ready to be executed (cf. section 11.5).

11.4.4. Error messages.

The first 6 translation passes perform a thorough checking of the formal correctness of the program. Every error found will be reported by a suitable message typed in red. An error message consists of the text

fejl i linie

(Danish for: error in line) followed by the number of the line where the error occurs and a short text characterizing the error. The line number is obtained by counting the CARRET symbols in the source program, line 0 being the one where the first begin appears. Line numbers may be obtained with the help of line output (cf. section 11.3.5).

When the translator has detected an error in the program the translation is discontinued after completion of pass 6 and the system returns to the OVERSÆTTER-KLAR-SITUATION. This means that every program is taken through the complete error detecting part of the translating process and that all errors of a program often will be detected in a single translation run.

Error messages are also produced when certain tables which are created by the compiler exceed the space allotted to them. In this case the OVERSÆTTER-KLAR-SITUATION will follow immediately.

Detailed explanations of the possible error messages and their meaning may be found in appendix 4.

11.5. KLAR-SITUATION.

On completion of compilation and when a new execution of a program is called following a termination of execution the message

klar

is typed and the machine will stop waiting for a character to be typed. If a SPACE is typed a normal run will take place. Other characters typed in this situation allow a choice of the units used for output, as explained in the following section.

11.6. CHOICE OF OUTPUT UNITS OR STOP RUN.

The running system allows a free choice of the output units associated with the standard output procedures (cf. section 8.1) or of a termination of the run. This choice must be made in the KLAR-SITUATION and may be repeated at any time during the run of the program. The choice is controlled by means of the control typewriter as follows:

Symbol typed	Danish clue	Meaning
a	alt til alle	All output will both be typed on the typewriter and punched on tape
s	skrivemaskine	All output will be typed. Nothing will be punched.
p	perforator	Nothing will be typed. All output will be punched.
Any symbol than a, s, p, or u.	other	skrv-output goes to type writer, tryk-output to punch.
u	ud	Stop run. The run will terminate with a slut-message.

When a new CHOICE OF OUTPUT UNITS OR STOP RUN is desired during the execution of a program the contents of the indicator register KA should be changed. This will cause a jump to new CHOICE OF OUTPUT UNIT OR STOP RUN to be made at the first following opportunity (usually within a few seconds). When the choice has been made the execution of the program is immediately continued unless u has been typed.

11.7. TERMINATION OF EXECUTION OF PROGRAM.

All regular runs of ALGOL programs terminate with a message. The possible terminating messages and their meaning are as follows:

slut — The program has passed through the final end of the program. (Slut is Danish for: end).

ak — The demand on storage space exceeds the capacity of the machine. This will be caused by having too many variables of any kind (simple or subscripted, labels, for statements, etc.) in action simultaneously. See section 10.2 (Ak is Danish for: alas).

array — The program tries to declare an array too large for the machine or one with a negative number of elements.

exp — The built-in procedure for calculating exp has been called with an argument which would cause the result to exceed the range of real variables (cf. section 7.3). This may also be caused by the operation ↑ with a real exponent.

index — A reference to a subscripted variable having subscripts outside the bounds of the corresponding declaration is made.

ln — The built-in procedure for calculating ln has been called with a negative argument. This may also be caused by calling the operation ↑ with an exponent of real type and a negative radicand.

spild — Arithmetic operation produces result outside the range of real variables (cf. section 7.3). The operation ↑ with integer exponent is first calculated with the absolute value of the exponent as exponent and may therefore cause spild even if the final result is 0.

sqrt — The built-in procedure for calculating sqrt has been called with a negative argument.

tromle ak — One of the standard procedures til tromle or fra tromle is called with a value of tromleplads outside of the permitted range (capacity of drum is exceeded, cf. section 10.5.3).

Following a terminating message the machine stops waiting for a control letter to be typed on the typewriter. If

k

is typed the system returns to the KLAR-SITUATION, ready for a new execution of the program (cf. section 11.5). Any other character will return the system to the OVERSÆTTER-KLAR-SITUATION (cf. section 11.3) ready for a new compilation, except for the case that the section of the drum which holds the compiler has been used for variables by the program just terminated (cf. section 10.5). If this is the case the message

væk

(Danish for: gone) is typed. It is then necessary to perform a new loading of the compiler into the machine (cf. section 11.1).

Appendix 1.

PASS INFORMATION.

The pass information is obtained as an optional output during translation (cf. section 11.3.6). It consists of the following:

At the end of pass 1, just before the epilogue (cf. section 11.3.4):

1. linie <number of the last line of the ALGOL program> end

Following each pass: two or three integers. The first of these always gives the number of drum tracks used to hold the intermediate output from the pass. The remaining have the following meaning:

Pass 1. Number of drum tracks reserved for long text strings.

Pass 2. a. 979 - the number of different identifiers in the program apart from standard identifiers (minimum 512).
b. 364 + number of words used for long identifiers.

Pass 3. The number of blocks in the program. (Max 1023).

Pass 4. a. The maximum depth in the stack used for collecting the declarations belonging to each block at the begin of the block and for rearranging procedure calls, rounded up to the nearest multiple of 10. (Max 512).
b. The maximum level of blocks.

Pass 5. a. The number of redeclarations of identifiers.
b. The number of occurrences of identifiers in the program apart from standard identifiers and the place where the identifier is declared.

Pass 6. a. The maximum of the number of words used in the operator stack. (Max. 50).
b. The maximum of the number of words used in the operand stack. (Max. 70).

Pass 7. Maximum number of words used in the stack of operand descriptions. (Max. 55).

Pass 8. 291 - the sum of maximum number of words used in the two program point stacks.

The number of drum tracks used by the finally translated program is the sum of the number of tracks reserved for long text strings (pass 1) and the number of output tracks from pass 8. These tracks are placed from track 319 and downwards as far as necessary.

Appendix 2.

PASS OUTPUT.

If desired the compiler will produce printed output of the internal output produced by each pass (cf. section 11.3.7). The following pages give the code for the basic structures of each output.

The output from the passes 1 to 7 has the form of a uniform sequence of integers in the range from 0 to 1023 printed with 10 in each line. Depending on the context these integers represent specific delimiters or are attached as parameters to adjecent integers to form structures of up to 6 integers. The syntax of these structures is specified fully. Some of the peculiarities of the order in which delimiters and parameters will appear in the output are caused by the fact that passes 4 and 8 proceed in the reverse direction. This causes the output from passes 3, 4, and 7 to be scanned by the following passes in the reverse direction of that in which it is produced and in which it appears in the printed output. The structures are everywhere described in the order in which they appear as output.

Pass 8 produces the final machine code. This is available as output from pass 9. The specifications given include only those instructions which refer to the fixed administration and which are therefore not comprehensible to those familiar with the GIER machine code.

As a further aid to the understanding of the pass output the purpose and function of each pass shall be given briefly as follows:

Pass 1. Conversion to reference language. Strings and layouts are assembled.

Pass 2. Identifier matching. Each distinct identifier will be associated with an integer from 512-1022.

Pass 3. Analysis and check of delimiter syntax. Delimiters of multiple meaning are replaced by distinctive characters. Extra delimiters are inserted to facilitate the later scanning.

Pass 4. Collection of declarations and specifications at the begin of each block. Rearrangement of procedure calls.

Pass 5. Distribution of declarations and specifications. Each identifier is replaced by its full description. Storage allocation of variables.

Pass 6. Conversion of expressions to Reverse Polish notation (see e.g. Computer Journal Vol 5, no. 3, 210). Type checking.

Pass 7. Generation of machine instructions for expressions. Allocation of working variables.

Pass 8. Final addressing of program. Segmentation into drum tracks. Production of final machine code.

Pass 9. Rearrangement of the program tracks on the drum.

Output unit	Meaning	Pass where used	Output unit	Meaning	Pass where used	Output unit	Meaning	Pass where used
1	a	2	41	M	2	131	array	3
2	b	2	42	N	2	135	switch	3
3	c	2	43	O	2	137	string	3
4	d	2	44	P	2	139	label	3
5	e	2	45	Q	2	141	value	3
6	f	2	46	R	2	151	;	3
7	g	2	47	S	2	156	end	3
8	h	2	48	T	2	160	else	3
9	i	2	49	U	2	165	(	3
10	j	2	50	V	2	167	then	3
11	k	2	51	W	2	169	do	3
12	l	2	52	X	2	174	:	3
13	m	2	53	Y	2	176	step	3
14	n	2	54	Z	2	178	until	3
15	o	2	55	Æ	2	180	while	3
16	p	2	56	Ø	2	182	]	3
17	q	2	57	0	2,3	190	[	3
18	r	2	58	1	2,3	200	,	3
19	s	2	59	2	2,3	207	:=	3
20	t	2	60	3	2,3	211	)	3
21	u	2	61	4	2,3	225	true	3
22	v	2	62	5	2,3	226	false	3
23	w	2	63	6	2,3	227	×	3
24	x	2	64	7	2,3	228	/	3
25	y	2	65	8	2,3	229	∧	3
26	z	2	66	9	2,3	230	:	3
27	æ	2	67	.	3	231	≤	3
28	ø	2	68	₁₀	3	232	<	3
29	A	2	72	+	3	233	=	3
30	B	2	76	-	3	234	≥	3
31	C	2	85	-,	3	235	>	3
32	D	2	87	go to	3	236	≠	3
33	E	2	91	begin	3	237	∧	3
34	F	2	93	for	3	238	∨	3
35	G	2	100	if	3	239	≡	3
36	H	2	102	own	3	240	⇒	3
37	I	2	107	integer	3	1021	end pass	2
38	J	2	112	real	3	1022	<4 bytes>	
39	K	2	117	boolean	3		lit string	2
40	L	2	124	procedure	3	1023	CAR RET	2

<4 bytes> ::= <text on drum>|<short text>|<layout>
<text on drum> ::= 0 <track relative> 0 <track number>
<layout> ::= <layout bits 0-9><layout bits 10-19><layout bits 20-29>
<layout bits 30-39>
Output unit 1021 (end pass) will appear in the following context:
. . . . . .
linie <line number of last line> end
156 <epilogue>; 1021 0
<number of output tracks><number of tracks for strings>

PACKING OF LAYOUTS AND STRINGS.

Layouts. These are packed in one word as follows:

Bits 0 - 19 A 1 in position p indicates that character number p in the layout (not counting SPACEs) is followed by SPACE.
- 20 - 23 b = number of significant digits
- 24 - 27 h = - - digits before the point
- 28 - 29 fn = sign of number part (no sign = 0, - = 1, + = 2, ± = 3)
- 30 - 33 d = number of digits after the point
- 34 n, 0 if no n, 1 if n
- 35 - 37 s = number of digits in exponent
- 38 - 39 fe = sign of exponent (code as for fn)

Other strings. These are packed character by character. One character uses 6 bits. The numerical value of the character is the one given in section 6.5 of the Manual with the exception of CAR RET which is represented by 63. Characters for UPPER CASE and LOWER CASE are included as needed, but all strings are understood to begin and end in lower case. The end of a string is indicated by the character value 10. The strings having 6 or fewer characters are packed in one word and carried through the translation process like numbers. Longer strings are stored on the drum during pass 1 and are represented during translation and at run time by a word referring to the drum.

Packing of short strings (6 or fewer characters):

Bits 0 - 3 The constant 10
- 4 - 9 Character no. 6 -
- 10 - 15 - - 5 |
- 16 - 21 - - 4 Unused character positions are
- 22 - 27 - - 3 set to 10
- 28 - 33 - - 2 |
- 34 - 39 - - 1 -

The word referring to a long string has the following structure:

Bits 0 - 9 The constant 0
- 10 - 19 track relative address, tr
- 20 - 29 The constant 0
- 30 - 39 track number, tn

On the drum the characters are stored in consecutive words on track tn in relative addresses tr, tr+1, tr+2, ... etc. The word following the one having relative address 39 on track tn is word 0 on track tn-1. Within each word the characters are packed in the following order:

Bits 0 - 5 Character no. 7
- 6 - 11 - - 6
- 12 - 17 - - 5
- 18 - 23 - - 4
- 24 - 29 - - 3
- 30 - 35 - - 2
- 36 - 41 - - 1 (bit 40 is mark a, bit 41 is mark b)

Appendix 2. PASS OUTPUT: PASS 2.

Output unit	Meaning	Pass where used
57	0	3
58	1	3
59	2	3
60	3	3
61	4	3
62	5	3
63	6	3
64	7	3
65	8	3
66	9	3
67	.	3
68	␣	3
72	+	3
76	-	3
77	CARRET	3
78<4 bytes>	lit string	3
85	-,	3
87	go to	3
91	begin	3
93	for	3
100	if	3
102	own	3
107	integer	3
112	real	3
117	boolean	3
124	procedure	3
131	array	3
135	switch	3
137	string	3
139	label	3
141	value	3
151	;	3
156	end	3
160	else	3
165	(	3
167	then	3
169	do	3

Output unit	Meaning	Pass where used
174	:	3
176	step	3
178	until	3
180	while	3
182	]	3
190	[	3
200	,	3
207	:=	3
211	)	3
225	true	3
226	false	3
227	×	3
228	/	3
229	↑	3
230	:	3
231	≤	3
232	≤	3
233	=	3
234	>	3
235	≥	3
236	≠	3
237	∧	3
238	∨	3
239	=	3
240	=>	3
241	end pass	3
512-979	Free identifiers	5
980	lyn	6
981	tromleplads	5
982	taststreng	6
983	læstegn	6
984	tasttegn	6
985	sættegn	6
986	tegn	6
987	tast	6
988	til tromle	6
989	fra tromle	6

Output unit	Meaning	Pass where used
990	tryktegn	6
991	skrvtegn	6
992	trykvr	6
993	tryktom	6
994	tryktekst	6
995	tryktab	6
996	tryksum	6
997	trykstop	6
998	trykslut	6
999	trykml	6
1000	trykkopi	6
1001	trykklar	6
1002	trykende	6
1003	tryk	6
1004	streng	6
1005	sqrt	6
1006	skrvvr	6
1007	skrvtekst	6
1008	skrvtab	6
1009	skrvml	6
1010	skrvkopi	6
1011	skrv	6
1012	sin	6
1013	sign	6
1014	læst	6
1015	læsstreng	6
1016	læs	6
1017	ln	6
1018	exp	6
1019	entier	6
1020	cos	6
1021	arctan	6
1022	abs	6

Output unit 241 (end pass) will be followed by:
1) Smallest output value used for identifiers - 1, (= min identifier)
2) 0.
Example:

```
. . . . . 241  949  0
```

<4 bytes> is defined in the output from pass 1.

Output values 0-168 and 512-1022 are processed in pass 4, the remaining in pass 6.

Output unit	Meaning	Output unit	Meaning	Output unit	Meaning
0	CAR RET	76	spec string	211	end go to
<4 bytes>4	lit string	80	value	212	for
<number> 5	lit integer	84	formal	213	step
<number> 6	lit real	88	switch list	214	until
<log val>7	lit bool	92	end call	215	end do
16	switch param	96	end clean	216	pos
20	call param	100	end block	217	neg
24	beg call	104	end proc	218	+
28	beg func	108	bounds	219	-
32	decl switch	112	bound colon	220	×
37	decl array intg	116	begin	221	/
38	decl array real	120	;	222	:
39	decl array bool	124	do	223	↑
40	decl proc	128	thenst	224	<
41	decl proc intg	132	elsest	225	≤
42	decl proc real	136	trouble	226	=
43	decl proc bool	140	:= for	227	≥
45	decl simple intg	144	simple for	228	>
46	decl simple real	148	step elem	229	≠
47	decl simple bool	152	while elem	230	∧
48	decl label	156	while	231	∨
53	decl own intg	160	end assign	232	=>
54	decl own real	164	:=	233	≡
55	decl own bool	168	first :=	234	⌈
56	spec switch			235	[
61	spec array intg	200	bound comma	236	]
62	spec array real	201	subscr comma	237	)
63	spec array bool	202	proc ;	238	-,
64	spec proc	203	ifex	239	simple for do
65	spec proc intg	204	ifst	240	step element do
66	spec proc real	205	thenex	241	while elem do
67	spec proc bool	206	elseex	512-979	Free identifiers
69	spec simple intg	207	delete call		
70	spec simple real	208	end elseex	980-1022	Std.identifiers
71	spec simple bool	209	end elsest		See output from pass 2
72	spec label	210	end thenst		

<4 bytes> ::= <text on drum>|<byte><byte><byte><byte>
<text on drum> ::= 0 <track relative> 0 <track number>
<number> ::= <bits 0-9><bits 10-19><bits 20-29><bits 30-39>
<logical value> ::= <true>|<false>
<true> ::= 0 256 0 0
<false>::= 1023 512 0 0

Following the normal output two output units appear:
1. Smallest output unit used for identifiers - 1, (= min identifier)
2. 0

Appendix 2. PASS OUTPUT: PASS 4.

Pass 4 is a reverse pass. Consequently the output bytes appearing first will refer to the last part of the program et vice versa.

Where 3 output byte values are given together they refer to the types integer, real, Boolean, in this order.

Output unit	Meaning
0	CAR RET
<constant>	
- 4	lit string
- 5, 6, 7	lit
<base points>	
- 16	begin block
- 20	begin procedure
<line identifier>	
- 24	proc value
- 28	end pass
<line identifier><no.of elem>	
- - 32	decl switch
<identifier list><no.of subsc>	
- - 37,38,39	declare array
<line identifier><no.of param>	
- - 40	declare procedure
- - 41,42,43	decl type proc
<identifier list>	
- 45,46,47	decl simple
- 48	declare label
- 53,54,55	declare own
- 56	specify switch
- 61,62,63	specify array
- 64	specify procedure
- 65,66,67	specify type proc
- 69,70,71	specify simple
- 72	specify label
- 76	specify string
- 80	value
- 84	formal
- 88	switch list
<identifier><no.of parameters>	
- - 92	begin call
96	begin local
<no.of ident.><adr. of coef>	
- -100	bounds
104	end block
<identifier>	
- 108	end procedure

Output unit	Meaning
185	expression
186	end call
187	begin expr
200	bound comma
201	subscr comma
202	proc ;
203	ifex
204	ifst
205	thenex
206	elseex
207	delete call
208	end elseex
209	end elsest
210	end thenst
211	end go to
212	for
213	step
214	until
215	end do
216	pos
217	neg
218	+
219	-
220	×
221	/
222	:
223	↑
224	<
225	≤
226	=
227	>
228	≥
229	≠
230	∧
231	∨
232	=>
233	=
234	(
235	[

Output unit	Meaning
236	]
237	)
238	-,
239	simle for and do
240	step element do
241	while elem and do
242	end local
243	bypass label
244	go to bypass label
245	label colon
246	bound colon
247	end bounds
248	end expression
249	end design expr.
250	then statement
251	else statement
252	:= for
253	step element
254	first:=
255	while
256	while element
257	prepare assign
258	simple for
259	end assign
260	do
261	while label
262	:=
512- 979	Free identifiers
980- 1022	Standard identifiers as from pass 2

<constant> ::= <bits 30-39><bits 20-29><bits 10-19><bits 0-9>
<base points> ::= <base address for variables><base address for working>
<base address for working> ::= 1024 - number of locations used for
local variables - number of locations used for local program points
<base address for variables> ::= 1024 - number of locations used
for local program points
<identifier> ::= <byte value in range from 512 to 1022>
<line identifier> ::= <line number><identifier>
<identifier list> ::= <line identifier>|
<identifier list><line identifier>

The last bytes appearing in the output refer to the outermost block and the entire program and are as follows:
1.2. <base points>
3. 1021 - maximum block number - number of owns
4. 2 + number of owns
5. smallest output unit used for identifiers - 1
6. 0

STORAGE ALLOCATION OF QUANTITIES IN THE STACK.

The storage arrangement of variables and similar quantities is formed during passes 4 and 5. Each ALGOL block forms an independent arrangement and the storage allocation results in the attachment of a block number and a block relative address to each quantity. The block number counts the level of nesting of blocks, the outermost block of the program being block number 0. The block relative addresses reflect the order in which the locations are placed within the section of the stack reserved when the block is activated at run time. With a view to the manner in which the locations will be reserved during program execution 3 groups of quantities are distinguished. Within each group locations are assigned to the quantities of the program in the order in which these quantities appear in the given ALGOL text. The 3 groups and the storage requirements of each item within them are as follows:

Group 1: Locations which have been initialized before the entry into the local declaration. This group exists only for procedure body blocks and is at run time reserved and initialized by the procedure call. Reservations: return information (1 location at relative address = 2), followed by formal locations (1 for each formal parameter at relative addresses 3, 4, ...).

Group 2: Locations which are initialized by the local declaration. One location is used for each program point (entry to labelled point or procedure body). A switch declaration will use one location for the switch identifier followed by one location for each switch element. The body of a type procedure will need one location for the procedure value, at relative address -1 (= 1023). The block information will need 2 locations, always at relative addresses 0 and 1. All other locations within the group have negative relative addresses, the first of these being the so-called base address for variables.

Continued on page 49

```
150 <const>                                  literal string
151,152,153 <const>                          literal integer, real, Boolean
154,155,156 0 <block adr>                    own integer, real, Boolean
157 <no.of elements><block adr>              switch
158,159,160 <no.of subscripts><block adr.>   array, integer, real, Boolean
161,162,163,164 <no.of param.><block adr.>   proc, no type, int, real, Bool
165,166,167      0 <block adr.>              simple integer, real, Boolean
168              0 <block adr.>              label
169,170,171      0 <block adr.>              formal array, integer, real, Boolean
172,173,174,175  0 <block adr.>              formal proc, no type, int, real, Bool
176,177,178      0 <block adr.>              formal simple, int, real, Bool
179              0 <block adr.>              formal label
180              0 <block adr.>              formal switch
181              0 <block adr.>              formal string
182,183,184 <no.of param><block adr.>        procedure call or assign,
                                               integer, real, Boolean
185       expression
186       end call
187       begin expression
188 0 0 0 undeclared
200 bound comma
201 subscr comma
202 proc ;
203 ifex
204 ifst
205 thenex
206 elseex
207 delete call
208 end elseex
209 end elsest
210 end thenst
211 end go to
212 for
213 step
214 until
215 end do
216 pos
217 neg
218 +
219 -
220 ×
221 /
222 :
223 ↑
224 <
225 ≤
226 =
227 ≥
228 >
229 ≠
230 ∧
231 ∨
232 =>
233 ≡
234 ¬
235 [
236 ]
237 )
238 -,
239 simple for and do
240 step element and do
241 while elem and do
242 end local
243 bypasslabel
244 goto bypasslabel
245 label colon
246 bound colon
247 end bounds
248 end ex
249 end des
250 then st
251 else st
252 := for
253 step elem
254 first:=
255 while
256 while element
257 prepare assign
258 simple for
259 end assign
260 do
261 while label
262 :=
263 <number of elements>
      local switch
264 <number of parameters>
      local procedure
265 local label
266 blind
267 end pass
268 CAR RET
269 <working base>
      begin block
270 <working base>
      begin procedure
271 <identifier>
      end procedure
272 <relative addr - 2>
      take value unrounded
273 <relative addr - 2>
      take value rounded
274 begin local
275 <no.of actuals>
    <identifier>
      begin call
276 end block
277 <rel.addr.of coeff>
    <no.of arrays>
      bounds
980-1022 Std.identifiers
```

<constant> ::= <bits 0-9><bits 10-19><bits 20-29><bits 30-39>
<block adr.> ::= <block relative address><block number>
<identifier> ::= <standard procedure identifier byte|<identifier description starting with byte in range from 154 to 184 (see above)>
First byte in output = 1021 - max. block number - number of owns
Second byte in output = working base for outermost block.

Continued from page 47.

Group 3: Locations reserved, but not initialized, by the local declaration. Simple variables use one location each, irrespective of their type. An array segment uses one location for each array identifier followed by (1 + number of subscripts) locations for storage mapping coefficients. The relative addresses of the locations of this group are even more negative than those of group 2, the first being the so-called base address for working. Working locations created during pass 7 also belong to this group. They are placed at (base address for working - 1), (base address for working - 2), etc.

Example of relative addresses of a block.

```
real procedure P(f1, f2); real f1, f2;
begin switch s:= L1, L2;
      array a, b[1:n];
      real r;
      procedure Q; ;
L2:
L1:
end;
```

	Relative address		
	....	Working locations -	
	1011	(pass 7)	
base address for working:	1012	a	
	1013	b	Group 3
	1014	coeff. 1	
	1015	- 2	
	1016	r	
base address for variables:	1017	s	
	1018	element 1	
	1019	- 2	
	1020	Q	
	1021	L2	Group 2
	1022	L1	
	1023	value of P	
	0	block information	
	1	- - -	
	2	return - -	
	3	f1	Group 1
	4	f2	

Expressions are written in reverse Polish notation. Output values 81-90 are used for parameters in procedure calls and local declarations.

0 begin local
1 end local
<no of parameters + 1>
2 - begin call
<working base>
3 - begin procedure
<no of formals>
4 - end procedure
5 - end type procedure
<working base>
6 - begin block
7 end block
8 go to bypasslabel
9 label declaration
10 while label
11 begin expression
12 if
13 else statement
14 end else statement
15 for
16 := for
17 simple for
18 simple for and do
19 while
20 while element
21 while element and do
22 step
23 until
24 step element
25 step element and do
26 end do
27 end subscripts
28 subscript comma
29 end switch designator
30 round
31 abs
<track rel><track no>
32- -std.proc. 1 array par
33 entier
34 -,
35 negative
36 end RF expression
37 end R expression
38 end address expr
39 proc;
40 else RF expression
41 else R expression
42 else address expr
43 end else RF expr
44 end else R expr
45 end else addr expr
46 then
47 prepare assign
48 not used
49 :=
50 go to
51 +
52 -
53 ×
54 /
55 :
56 ↑ integer
57 ↑ real
58 < then
59 ≤ then
60 = then
61 ≥ then
62 > then
63 ≠ then
64 <
65 ≤
66 =
67 ≥
68 >
69 ≠
70 ∧
71 ∨
72 ≡
<block rel><bl.no>
73- -array
74- -simple variable
75- -own
76- -switch
77- -label
78- -formal
79- -procedure
80 address
<no. of elements>
81 - local switch
<no of parameters>
82 - local procedure
83 local label
84 procedure value
85 expression as param
<block rel><bl no>
86- -simple or label
87- -own parameter
88- -described in stack
<track rel><track no>
89- -std.proc.parameter
<constant>
90 - constant parameter
<track rel><track no>
91- -std.proc. no param
92- -std.proc. 1 RF par
93- -std.proc. 1 R par
94 prepare function call
<constant>
95 - constant
<block rel><bl.no>
96- -end proc call
<track rel><track no>
97- -end standard call
<rel.adr.coef><no ident>
98- -bounds
99 bound comma
100 end bounds
<block rel - 2>
101 - take value rounded
102 - take value unrounded
103 end pass
104 bypass label
105 take nonsense
106 trykende
107 trykslut
108 trykstop
109 tryktab
110 trykvr
111 lyn

First symbols in output:
1. Initial stack reference = 1021 - max. block number - no. of owns
2. Working base for outermost block = 1024 - number of words needed for program points etc. - number of words used for variables.

OPERAND ADDRESSES IN THE OUTPUT FROM PASS 7.

In the output from pass 7 references to variables in the stack (i.e. operand addresses) have been finally differentiated into the following classes:

1. Variables in the outermost block of program. The block number, which is 0, is omitted from the output. The final machine instructions, formed in pass 8, use absolute addresses.

2. Variables in the currently local block. Again the block number is unnecessary. The final machine addresses are p-relative; at run time the p-register always holds the stack reference of the currently local block.

3. Variables in intermediate blocks. These require both the block number and the relative address. In the final machine program s-relative addresses are used and the appropriate stack references are placed in the s-register by means of explicit ps-instructions referring to the DISPLAY. These ps-instructions are inserted as needed during pass 8.

In addition to the above three classes the following operands appear explicitly in the output from pass 7:

4. UA, the Universal Address. This is used for subscripted variables and variables called by name. The basic administrations for referring to these two kinds of entities place the absolute machine address of the relevant location in UA (the THUNK idea, see Ingerman, Comm. ACM. Jan.1961, 55).

5. UV, the universal value. Upon completion of a call of a type procedure the value of the function designator is found in UV.

```
<lit>              1 constant
<block relative>   2 absolute address (block 0)
<block relative>   3 p relative (in local block)
<block relative>   4 (p)-rel. (ind., local block)
<st.ad>            5 s-relative (block relative)
                   6 (UA) (ind., univ. address)
                   7 UV (univ. value)
10 if R<0 then R:=-R
11 R:= -R
12 procedure value
13 end UV expr
14 R:= round(RF)
15 end RF exp
16 end R expr
17 end address expr
18 R:= -, R
19 R:= R - epsilon
20 if R≠0 then R:= -1
21 RF:= store[UA]
22 RF:= RF - 0.5
23 RF:= float(R)
24 goto non loc. in s
25 goto computed
26 goto local in s
27 M:= UA
28 UA:= s
<op> 29 RF:=abs(<op>)
<op> 30 RF:= -<op>
<op> 31 RF:=RF + <op>
<op> 32 RF:= RF×<op>
<op> 33 R:= R ∧ <op>
<op> 34 R:= R ∨ <op>
<op> 35 R:= R = <op>
<op> 36 RF:=RF - <op>
<op> 37 RF:= RF/<op>
<op> 38 R:=-abs(<op>)
<op> 39 R:= abs(<op>)
<op> 40 RF:= <op>
<op> 41 R:= <op>
<op> 42 <op>:= RF
<op> 43 <op>:= R
<op> 44 <op>:= 0
<op> 45 M:= <op>
<op> 46 <op>:= M
<op> 47 s:= <op>

<op>   48 grnf V LA
<op>   49 acn MA
<op>   50 <op>:=RF marked
<op>   51 <op>:= 0 marked
<op>   52 <op>:= M marked
<op>   53 s:= adr(<op>)
<op>   54 mult 1st subscr
<op>   55 call formal
       56 integer divide
       57    -      -
       58    -      -
<op>   59 index call
<op>   60 switch call
       61 c54:= RF
       62 c68:= RF
       63 c54:= M
       64 c68:= M
       65 M:= store[UA]
       66 not input
<char> 67 punch <char>
<rel>  68 working:= R
<rel>  69 working:= RF
<rel>  70 working:= M
<subsc>71 mult next subsc
<rel>  72 end do
<rel>  73 end single do
       74 Radr:= input
       75 R00:= R0
<elem> 76 local switch
<lit>  77 const parameter
<-ap>  78 beg call
<tr.p> 79 end call std.
<tr.p> 80 std.proc.par
<st.ad>81 simple as par
<st.ad>82 simple bl. 0 par
<st.ad>83 desc. in stack
<st.ad>84 described in
          stack bl. 0

<st.ad>85 endproccall
       86 RF:= Radr
<tr.p> 87 std.proc no par
<tr.p> 88 std.proc 1 par
<tr.p> 89 std.func 1R
<-ap>  90 endproc
<-ap>  91 end type proc
<-ap>  92 end block
<bl.no>93 begin local
       94 end else
       95 bypass label
       96 label declar.
       97 begin expr
       98 begin proc
       99 do
      100 if
      101 for
      102 for label
      103 call ln
      104 else
      105 then
      106 if R>0 then goto
      107 if R<0 then goto
      108 if R=0 then goto
      109 if R≠0 then goto
      110 do NT
      111 do abs
      112 if R<0 then goto
      113 goto bypasslabel
      114 bypassabs
      115 prog point par
      116 (internal)
      117 (internal)
      118 endlocal
      119 return inf
<rel> 120 take for label
      121 RF:= nonsense
      122 call exp
      123 end pass
      124 complete array
      125 take last used
<op>  126 R:= array ref
<op>  127 <op>:= array id
<op>  128 reserve space
<ap>  129 s:= p + <ap>
      130 call integer
```

<op> ::= <operand description, see byte values 1 to 7 above>
<literal (lit)> ::= <bits 30-39><bits 20-29><bits 10-19><bits 0-9>
<stack address (st.ad)> ::= <block relative><block number>
<character (char)> ::= <value of character, see section 6.5>
<rel> ::= <relative in local block>
<subsc> ::= <subscript number - 1>
<elem> ::= <number of switch elements>
<-appetite (-ap)> ::= <1024 - number of locations reserved in stack>
<appetite (ap)> ::= <number of locations reserved in stack>
<track point (tr.p)> ::= <track relative><track number>
<block number (bl.no)> ::= <block number>

OUTPUT FROM PASS 8: FINAL MACHINE CODE.

Adressing.

Pass 8 generates the final machine program and places it in correct relative locations on the drum tracks. Pass 9 only rearranges the complete tracks on the drum. The final machine program is available as output from pass 9 and will be printed as normal program text. Each track will be headed with the track number. The first few locations on a track usually are occupied by literal constants needed by the program stored on the track. These will be referenced by r-relative addresses from the program. Other operands are addressed as in the output from pass 7. Jumps within the same track use r-relative addressing. All other program references specify the track relative address and the track number. The program starts in some unspecified location on the first track printed.

The following explanations only deal with those program parts which refer to the running system (the fixed administration placed in the cores from 835-1023). It should be noted that the absolute addresses given are subject to change if the running system is changed in any way in new editions of the compiler.

Parameters in local declarations and procedure calls.

The local declaration and the procedure call both serve to reserve and initialize locations in the stack. In the machine code they are represented by a call of the administration followed by one or more words which serve as parameters. The administration will take the parameters one by one and usually generate the contents of a location in the stack from the information given in the parameter. The parameters appear in the machine code in the reverse order of that in which they appear in the original ALGOL program.

Absolute addresses in a program having the maximum block nesting n and p own variables.

```
....        Variables in
-193-n-p        outermost block
-192-n-p    Block information of
-191-n-p       outermost block
-190-n-p    own no. 1
. . . .
-191-n      own no. p
-190-n      internal reservation
-189-n      DISPLAY [n]
. . . .
-190        DISPLAY [1]
-189  c1    DISPLAY [0]
```

-188 c33=c35. (1) mt c33: change sign. (2) c35 contains 0.5 floating, used for entier: srf c35, and : : srf c35 V NT; arf c35;

-187 c34. epsilon. sr c34 is used in certain relations.

-186 c37. UV, the universal value.

-185 c40. Floating 1.0 and true.

-184 c41. - -1.0 and false.

-183 c43. Nonsense, the value of output procedures.

-182 c46. last used in the stack. (1) arn c46, ck 10: form array identifier. (2) ps (c46): used for take value (when values are taken the formals are addressed relatively to last used).

-173 c68. Working location. Used for the exponent when calling ↑integer.

-172 c69. tromleplads.

-116 c2. hs c2, qq track relative + track number.29: transfer control.

-114 c3. hv(f) c3: go to local label referenced in the s-register.

-113 c42. hs c42, qq track relative + track number.29: transfer in parameter list (see about parameters below).

-112 c21. hv c21, qq track relative + track number.29: (1) Call standard procedure. (2) End local declaration.

- 98 c5. (1) hs c5, qq DISPLAY ref - appetite.29: begin local declaration. (2) hsf c5, qq - appetite.29: begin procedure call.

- 79 c26. pm array identifier, hs c26: index call, places the address of the subscripted variable in UA.

- 72 c36. UA, the universal address.

- 71 c20. hhf c20, qqf number of elements: switch in local declaration.

- 60 c54. Working location. Used for the radicand when calling ↑integer.

- 56 c50. qq p-relative addr. of length, hs c50: reserve space for array.

- 44 c55. hh c55: termination of execution.

- 42 c38. pm switch identifier, hs c38: take value of switch designator.

- 34 c22. pm formal identifier, hs c22: take formal, address to UA.

- 30 c9. hsn c9, qq track relative + track number.29: call standard procedure with no parameters.

- 20 c10. hs c10, qq track relative + track number.29: call standard procedure with 1 RF parameter.

- 19 c11. hs c11, qq track relative + track number.29: call standard procedure with 1 R parameter.

```
- 18 c7.   hh(f) c7: end type procedure.
- 16 c6.   hh(f) c6: end procedure.
- 14 c18.  arn(c36), hs c18: go to computed label.
- 13 c16.  hh(f) c16: go to non-local label.
- 10 c13.  hv(f) c13: end RF expression.
-  9 c14.  hv(f) c14: end R expression.
-  8 c12.  hv(f) c12: end UV expression.
-  7 c15.  hv(f) c15: end address expression.
-  4 c8:   , hs(f) c8: end block.
```

Parameter formats used only in local declarations.

hs-98 [=c5], qq DISPLAY ref. -<ap>.29:	begin local declaration
hhf-71 [=c20], qqf no. of elements:	local switch

Parameter formats used only in procedure calls.

hsf-98 [=c5], qqf - appetite.29:	begin call
ps (DISPLAY ref), pm s<block rel.>:	described in stack: array, switch, procedure identifier, formal name
qq DISPLAY ref, pm <absolute address>:	described in stack block 0
ps (DISPLAY ref), psn s <block relative>:	call declared procedure (end call).

Parameter formats used both in local declarations and procedure calls.

Any f-marked full word:	constant, in local decl. used only for the procedure value
qq 0 ,qq <track rel>+<track no>.29:	program point, left
qqf 0 ,qqf<track rel>+<track no>.29:	program point, right. Used in local declaration for: labels, procedures, expressions as switch elements. In procedure call: expressions, std. proc. identifiers, return information
psf (DISPLAY ref), psf s <block rel>:	simple var., label. In local decl.: switch elements.
qqf DISPLAY ref, psf <absolute adr>:	simple var., label, in block 0
hs-113 [=c42], qq<track rel>+<track no>.29:	track completed
hv-112 [=c21], qq<track rel>+<track no>.29:	continue in program, left
hvf-112 [=c21], qq<track rel>+<track no>.29:	- - - , right local decl: end local, Proc. call: call standard procedure.

Appendix 3. SELECTED EXECUTION TIMES.

The execution time of a program in GIER ALGOL depends not only on its individual algorithmic constituents, but also on the loop structure and the number of variables declared at the time when each part of the program is executed (cf. section 10.4). The times given below are based on actual timings at the machine and include an average track administration time such as it may be expected in loops which may be accomodated completely in the core store. Substantially longer execution times will result under the following circumstances: a) Frequent transfers of program tracks from drum are necessary (cf. section 10.4); b) A major part of the execution time of the program is spent in a loop with a cycle time of the order of 2 millisecond or less and this loop happens to have been placed across a program track transition by the compiler. A program suffering from the latter of these calamities may be cured by insertion of a suitable amount of neutral program (r:= r or the like) before the final end.

Algorithmic entity	Example	Execution time, milliseconds
Addition	a + b	0.12
Multiplication	a × b	0.18
Division	a / b	0.21
Square	a ↑ 2	0.18
Cube	a ↑ 3	0.4
Power, integer exponent	a ↑ i	
abs (exponent) = 1		3.8
10		5.5
100		8
1 000		10
10 000		12
100 000		14
1 000 000		16
Power, real exponent	a ↑ r	12
Subscripted variable		
1 subscript	A[i]	0.9
2 subscripts	B[i, j]	1.2
3 -	C[i, j, k]	1.5
Step-until element, constant step and simple upper limit, each loop	step 1 until n	0.6
Block with simple variables	begin real a; end	2.0
Block with array declaration	begin array a[1:10]; end	3.6
Reference to formal parameter called by name. Actual parameter is		
simple		0.4
expression		3.2
array identifier		0.0
switch identifier		0.0
procedure identifier		0.0

Call of declared procedure		
having an empty procedure body		
No parameter	P;	4.7
1 parameter	Q(a);	5.3
2 parameters	R(a, b);	5.7
3 -	S(a, b, c);	6.3
Call of standard procedure		
abs	abs(x)	0.17
arctan	arctan(x)	6.6
cos	cos(x)	6.0
exp	exp(x)	5.8
ln	ln(x)	5.6
sign	sign(x)	3.2
sin	sin(x)	5.8
sqrt	sqrt(x)	6.2

For the general description, refer to section 11.4.4.

The pass number is typed as an integer from 1 to 8 followed by a point (.) at the beginning of the first error message belonging to the pass.

The line referred to in an error message will normally be the line in which the error occurs, but there are exceptions to this rule: a) A construction appearing near the beginning or ending of a line may have its line number changed by one unit. b) One of the error messages from pass 5 may supply a quite misleading line number (see below). c) Error messages from passes 7 and 8 will always refer to line 0.

PASSES 1 - 8.

for stort program — program too big

This indicates that the capacity of the drum has been exceeded by the demands of the program text. Remedy: Use a version of the compiler which leaves more space on the drum, if such a version is available.

PASS 1.

forbudt tegn — forbidden sign

A character to which no meaning is assigned appears on the input tape.

fejl i sammensat symbol — error in compound symbol

A string a characters which represents some of the first characters of a compound symbol (cf. section 7.1.2), but not the following ones, appears in the input.

fejl i parameter delimiter — error in parameter delimiter

The construction)<letter string> is not followed by :(

ukorrekt brug af comment — incorrect use of comment

The delimiter comment is not preceded by begin or ;

udefineret layout — undefined layout

The compound symbol ≮ is followed neither by < nor by a layout (cf. section 8.3.1)

PASS 2.

for mange identifikatorer — too many identifiers

The program uses too many different or long identifiers. Remedy: Use the block structure to reduce the number of different identifiers.

PASS 3.

delimiter mangler — delimiter missing

Two operands (i.e. identifiers, numbers, logical values, strings, or compound expressions within parentheses) follow each other. Examples:

7.3 sin(5) 4 true r.77 r≮<string>

ikke tilladt operand — inadmissible operand

a) An operand appears in a wrong context. Examples:
   7:= begin true;
b) An operand is missing. Example:
   a:= [i]

forkert delimiter — wrong delimiter

a) The delimiter structure is impossible. Examples:
   begin r/i:= if go to if for
b) Binary operator does not follow operand. Example:
   i:= ×r;

operand mangler — operand missing

Operand is missing at end of construction. Example:
r:= r/;

forkert afsluttet konstruktion — wrong completion of construction

Parentheses, brackets, or bracket-like structures do not match. Examples:
r[i) begin r:= a + b, p(i, r;

talfejl — number error

A construction which in its first symbols conforms to the syntax for numbers is not terminated correctly, or a number is too big for the capacity of the machine. Examples:
20.; $17._{10}-3$ $7_{10}170$

stack overløb — stack overflow

The nesting of begin's, parentheses, etc. exceeds the capacity of the compiler.

PASS 4.

stack overløb — stack overflow

The stack formed during the reverse pass 4 exceeds the available capacity. This stack is used to transfer the information about the type and kind of each identifier and of each switch element from the place where it is declared (for labels, where it labels a statement) to the begin of the block in which it is local, and the information about each actual parameter to the left parenthesis of the call.

PASS 5.

dobbelt declaration — double declaration

The same identifier is declared twice in the same block or appears twice in the same formal parameter list. Note that labels are considered to be declared as explained in section 4.1.3.

dobbelt specific. — double specification

The same identifier is specified twice in the same procedure declaration heading.

manglende declaration — missing declaration
An identifier is used at a place where it is not declared. The line number associated with this error message will be misleading in the following two cases: a) The identifier is an actual parameter. The line number will point to the line in which the left parenthesis of the call appears. b) The identifier is a switch element. The line number will point to the line which contains the begin of the block in which the switch is declared.

manglende specific. — missing specification
The specification of a formal parameter is missing.

manglende formal — missing formal
An identifier is specified, but does not appear in the formal parameter list.

forbudt value specific. — inadmissible value spec.
A formal parameter which according to the specification given cannot be called by value appears in a value part.

stak overløb — stack overflow
The list of the identifiers which are redeclared simultaneously exceeds the capacity of the compiler.

PASS 6.

array subscript fejl 705 — subscript error
The number of subscripts given in a subscripted variable does not match the corresponding array declaration.

procedure parameter fejl — procedure parameter error
An additional integer in the message distinguishes two variants of this error:
796: An identifier preceding immediately a left parenthesis, (, does not conform to the procedure call implied in the construction by being of wrong kind or having a wrong number of parameters.
846: A procedure identifier appears in a context not consistent with its declaration.

forkert type <error number> — wrong type
The number associated with this error message indicates from where in pass 6 the error program has been called. Note that these numbers depend on the way in which pass 6 is stored and may change slightly if pass 6 is modified in any way (this may happen if mistakes are found and corrected). A more detailed description of the error associated with each integer is given in the table below. In this table the description

<i op> ::= <inadmissible operand>

indicates an operand which has wrong type or kind in the given context. Note that expressions are regarded as operands. The examples assume the following declarations:

```
integer i; real r; Boolean b; array a1[1:10], a2[2:4, 4:6];
switch s:= L, L2; procedure p0;; procedure p1(f); real f;
```

Error number	Error constructions	Examples
578	+<i op>|-<i op>|x<i op>|/<i op>|↑<i op>	+s /L
584	:<i op>	:r
587	<Boolean operand>:= <i op>	b:= r
592	< <i op>|≤ <i op>|= <i op>|	= b
	> <i op>|≥ <i op>|≠ <i op>	
595	∧ <i op>|∨ <i op>|≡ <i op>|-,<i op>	∨ (i - 2)
598	<i op><binary operator><i op>	i ∨ a1 b = s
601	<real operand>:= <i op>	r:= s
606	<integer operand>:= <i op>	i:= p
618	abs(<i op>)|arctan(<i op>)|cos(<i op>)|	cos(a2) ln(r = i)
	entier(<i op>)|exp(<i op>)|ln(<i op>)|	
	sign(<i op>)|sin(<i op>)|skrvkopi(<i op>)|	
	skrvml(<i op>)|skrvtegn(<i op>)|sqrt(<i op>)|	
	streng(<i op>)|sættegn(<i op>)|	
	trykkopi(<i op>)|trykml(<i op>)|	
	tryktegn(<i op>)|tryktom(<i op>)	
632	go to <i op>|switch sw:= <i op>,	go to b
635	;<i op>;	; r ;
642	<i op><binary operator>	b = r∧ (i - 2)∨
648	<i op>[	i[
651	til tromle(<i op>)|fra tromle(<i op>)	til tromle(r)
659	then <i op>	then s
679	<i op> else <i op>	2 - r else b
	<i op> else if . . . then <i op>	
688	:= <i op>:=	r:= b:= b ∨ b
691	<i op> step|<i op> until|for . . . <i op>,|	i = r step
	for . . . <i op> do	
698	<i op>]	p0]
715	<i op>:=	p0:=
726	<i op> then|while <i op> do	if r then
729	for <i op>:=	for a1:=
733	Inadmissible subscript	a1[L] a1[i=r]
736	<i op> : (in array declaration)	array q[b:1];

PASS 7.

talfejl number error

An arithmetic expression having only numbers as operands results in a value outside the range of the machine. Examples:

1/0 $7_{10}35\times9.2_{10}135$

PASS 8

stak overløb stack overflow

The two stacks of program points used during pass 8 exceed the capacity of the compiler. Remedy: reduce the number of labels and of nested for and conditional statements used simultaneously.